The Dissertation Committee for Ikhwan Lee
certifies that this is the approved version of the following dissertation:

Fine-Grained Containment Domains for Throughput Processors

Committee:

Mattan Erez, Supervisor

Mike Parker

Keshav Pingali

Vijay Janapa Reddi

Nur A. Touba

Fine-Grained Containment Domains for Throughput Processors

by

Ikhwan Lee, B.S.E.; M.S.E.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2015

To my wife, Ji Heun Choi

Acknowledgments

During my journey to a PhD, I have received support and encouragement from a great number of individuals. This acknowledgment is my attempt at expressing my sincere gratitude to some of them.

First, I would like to thank my advisor, Mattan Erez, for the patient guidance and mentorship in the face of numerous obstacles. His immense knowledge and insight on computer architecture and beyond has challenged me throughout my research, teaching me what it takes to be a great researcher.

I am indebted to the other members of my dissertation committee: Dr. Mike Parker, Professor Keshav Pingali, Professor Vijay Janapa Reddi, and Professor Nur A. Touba. Their keen insights and comments helped me develop and elaborate the concepts in my dissertation.

I would also like to thank my fellow LPH group members – those who have moved on, those still struggling, and those just beginning – for their support, feedback, and friendship. My life as a graduate student would not have been as fun without all the discussions and pizzas we had together.

I had an opportunity to collaborate with the researchers at NVIDIA Research as a research intern. I would like to take this opportunity to express my gratitude to my manager Stephen Keckler, and to my mentors Brucek

Khailany and Mike Parker. Thanks to their support and valuable feedback, I was able to develop the internship work into this dissertation.

During my time at UT, I was fortunate to have built friendships that would last a lifetime. I am grateful to Min Kyu Jeong, Seung Yun Nam, Donghyuk Shin, Sangmin Lee, and Dam Sunwoo for being good friends. Life in Austin would have been very different without their friendship.

Finally, I would like to thank my family. My wife, Ji Heun Choi, made innumerable sacrifices to support me through this journey, and I could not have finished this dissertation without her. I am also thankful to my parents, Sang Yong Lee and Kyung Pil Kho, for their unconditional faith in me. It has always given me confidence when I was in doubt. I should also thank my parents-in-law, Jaeheun Choi and Boksoon Jun, for their love and support.

Ikhwan Lee

August 2015, Austin, TX

Fine-Grained Containment Domains for Throughput Processors

Publication No. _____

Ikhwan Lee, Ph.D.
The University of Texas at Austin, 2015

Supervisor: Mattan Erez

Continued scaling of semiconductor technology has made modern processors rely on large design margins to guarantee correct operation under worst case conditions. Design margins appear in the form of higher supply voltage or lower clock frequency, leading to inefficiency. In practice, it is rare to observe such worst-case conditions and the processor can run at a reduced voltage or higher frequency experiencing only few infrequent errors. Recent proposals have used hardware error detectors and recovery mechanisms to detect and recover from these rare errors, a technique known as timing speculation. While this is effective for out-of-order processors with inherent capability to recover from misspeculation, implementing similar hardware for throughput processors such as the Graphics Processing Units (GPUs) is prohibitively costly due to the massive amount of thread context that needs to be preserved. Furthermore, recovery overhead is much higher since the SIMD (Single Instruction

Multiple Data) execution model of GPUs require multiple threads to roll back together in case of an error.

In this dissertation, I develop a hardware/software co-design approach to enable reduced-margin operation on GPUs that overcomes the limitations of existing techniques. The proposed scheme leverages the hierarchical programming model of GPUs to provide hierarchical and uncoordinated local checkpoint-recovery. By decomposing a program into a hierarchically nested tree of code blocks which I refer to as *containment domains (CDs)*, the program becomes amenable to automatic analysis and tuning, and an optimum trade-off can be made between preservation and recovery overhead. To aid this optimization process, an analytical model is developed to estimate the performance efficiency of a given application setting at a given error rate. With the analytical model, an exhaustive search can be performed to find the optimal solution. The tunability also allows the proposed scheme to easily adapt to a wide range of error rates making it future proof against emerging uncertainties in semiconductor design.

The proposed scheme combines software and hardware components to achieve the highest efficiency in preservation, restoration, and recovery. The software components include: 1) an API and runtime that lets the programmers describe the hierarchy of containment domains within an application and preserve the state required for rollback recovery, and 2) a compiler analysis that automatically inserts preservation routines for register variables. The hardware components include: 1) a stack structure to keep track of recovery

program counters (PC), 2) a set of error containment mechanisms to guarantee that no erroneous data is propagated outside of a containment domain and 3) an error reporting architecture that keeps track of affected threads and initiate recovery of them.

Table of Contents

[1]The general concept of containment domains discussed in this chapter is based on the prior publication [1]. The general concept and semantics are developed through open discussions among all authors. Jinsuk Chung developed the analytical model, Ikhwan Lee defined the system parameters and error/fault model, and Michael Sullivan, Jee Ho Ryoo, and Dong Wan Kim mapped applications to containment domains.

List of Tables

List of Figures

Listings

Chapter 1

Introduction

Reliability and resilience are major design concerns in future processor design. The growing number of components and the decreasing inherent reliability of components in future fabrication technologies may result in error and fault rates that are orders of magnitude higher than those of today. Technology scaling may also make systems be more sensitive to various error modes such as thermal and voltage emergencies, leading to widely varying error rates. Existing resilience mechanisms, however, either are not designed for high error rates or are tailored to, and optimized for, a specific error mode occurring at a specific error rate. While they are effective for today's systems, they will not be able to efficiently handle growing error rates with wide dynamic range. In this dissertation, I develop a hardware/software cooperative mechanism that can handle wide range of error rates that are largely ignored by current resilience schemes, and show that this mechanism can enable timing speculation on throughput-oriented processors such as current Graphics Processing Units (GPUs) resulting in energy savings.

The proposed resilience scheme is based on the concept of *containment domains* (CDs) [1]. CDs are a programming construct that enables applica-

tions to express resilience needs and to interact with the system to tune and specialize error detection, state preservation and restoration, and recovery. With CDs, software can preserve and restore state in an optimal way within the storage hierarchy and can efficiently support uncoordinated recovery. CDs have weak transactional semantics and are designed to be nested to form a CD hierarchy. The core semantic of a CD is that all data generated within the CD must be checked for correctness before being communicated outside of the domain and that each CD provides some means of error recovery. Failures in an inner CD within the CD hierarchy are encapsulated and recovered by the domain in which they occur; therefore, they are *contained* without global coordination. A specific error may be too rare, costly, or unimportant to handle at a fine granularity. For this reason, an inner CD can escalate certain types of errors to its parent. This flexibility of expressing how and where to preserve and restore data, as well as how to recover from errors, allows CDs to perform well across wide range of error rates. A CD can be of any granularity ranging from a single instruction to an entire application. The proposed scheme focuses on dealing with high error rates and thus *fine-grained containment domains* (FGCDs) that exploit the on-chip storage hierarchy are introduced. FGCDs share the same concept and semantics with CDs, but further incorporate hardware and runtime support to efficiently implement preservation and recovery at a fine granularity.

This dissertation focuses on applying the idea of fine-grained containment domains to throughput-oriented processors, or more specifically, modern

GPUs. GPUs are widely being adopted as a general purpose computing platform and they are a suitable candidate for FGCDs because of their hierarchical programming and execution model. GPUs also lack speculative execution support, and thus existing architecture-level checkpoint-recovery mechanisms [2, 3] developed for out-of-order processors cannot be easily adopted, making FGCDs a more attractive solution to GPU resiliency. With fine-grained containment domains, a modern GPU can efficiently handle both high and widely varying error rates. This opens up an opportunity to adopt timing speculation on GPUs. Timing speculation is a technique to run a circuit at a faster clock (or lower voltage) than safely allowed by the design. In other words, it is *speculated* that the circuit timing will be met for the vast majority of time, and in case of rare misspeculation (or timing error), the error is detected and the circuit is recovered to a correct state. The energy efficiency is optimized by balancing the cost of error recovery and the benefit of faster clock frequency (or lower voltage). Timing speculation is especially promising for future fabrication technologies where large design margins are required to guarantee correct operation, and fine-grained containment domains opens up the opportunity to take advantage of the idea for GPUs.

1.1 Thesis Statement

A resilient GPU design incorporating the concept of fine-grained containment domains can provide higher energy-efficiency by allowing the proces-

sor to run at more energy optimal operating points in terms of voltage and frequency.

1.2 Contributions

The main focus of this dissertation is on improving the energy efficiency of throughput-oriented processors (or GPUs) by employing the idea of timing speculation. In order to enable timing speculation on GPUs, *fine-grained containment domain*, which is a hardware/software cooperative mechanism to perform fine-grained state preservation, restoration, and recovery on GPUs, is designed and implemented. With FGCDs, energy optimal operating points for various General Purpose GPU (GPGPU) applications are found and the corresponding energy savings are reported.

More specifically, the main contributions this dissertation are:

1. I develop fine-grained containment domains for throughput-oriented processors such as current GPUs. FGCDs are a hardware/software cooperative mechanism to perform fine-grained and hierarchical state preservation, restoration, and recovery. FGCD consists of 1) an API that allows the programmer to tune the location and frequency of preservation, 2) a runtime that implements the services specified by the API, and 3) a set of architecture support to aid error reporting and error recovery.

2. I build an analytical model to estimate the efficiency of an application for a given FGCD mapping at a given error rate. FGCD mapping of

an application determines the location and frequency of preservation as well as the grouping of threads for coordinated error recovery. Analytical model enables fast evaluation of various FGCD mappings at varying error rates and derives an optimal FGCD mapping and operating point for each application.

3. I evaluate the applicability of timing speculation on GPUs by running cycle-based simulation of various GPGPU workloads. First, analytical model results are verified with simulation results, and optimal FGCD mappings and operating points derived by the analytical model is used to estimate the energy savings that can be achieved by FGCD.

1.3 Dissertation Organization

The remainder of this dissertation is organized as follows: Chapter 2 reviews necessary background information for understanding contemporary GPU architectures and programming model, explains the problem of design margins, and discusses previously proposed timing speculation schemes. Chapter 4 explains the semantics of fine-grained containment domains, develops the analytical model, and describes software and hardware components of FGCDs. Chapter 5 discusses the evaluation methodology used in the dissertation, and Chapter 6 presents a set of experimental results. Lastly, Chapter 7 concludes the dissertation.

Chapter 2

Background

This chapter provides basic background information for understanding contemporary GPU architectures and their programming and execution model. We also discuss the problem of design margins (guard-bands) in the future process technology and explain how the concept of better than worst-case design is applied to overcome the problem. We also discuss broader range of literature in the area of system resilience and explain how they relate to FGCDs proposed in this dissertation.

2.1 GPU Architecture

Figure 2.1 shows a high-level block diagram of a modern GPU architecture. Modern GPUs are composed of multiple single-instruction multiple-data (SIMD) processors referred to as *streaming multiprocessors* (SMs) by NVIDIA [4]. The underlying off-chip memory system is shared among SIMD processors through a last level (L2) cache. The SIMD-width is often wide (ranging from 16 to 64) to amortize the control overhead while achieving high throughput. SIMD lanes are simple in-order pipelines and each SIMD processor keeps a massive number of threads to hide the latencies of memory accesses

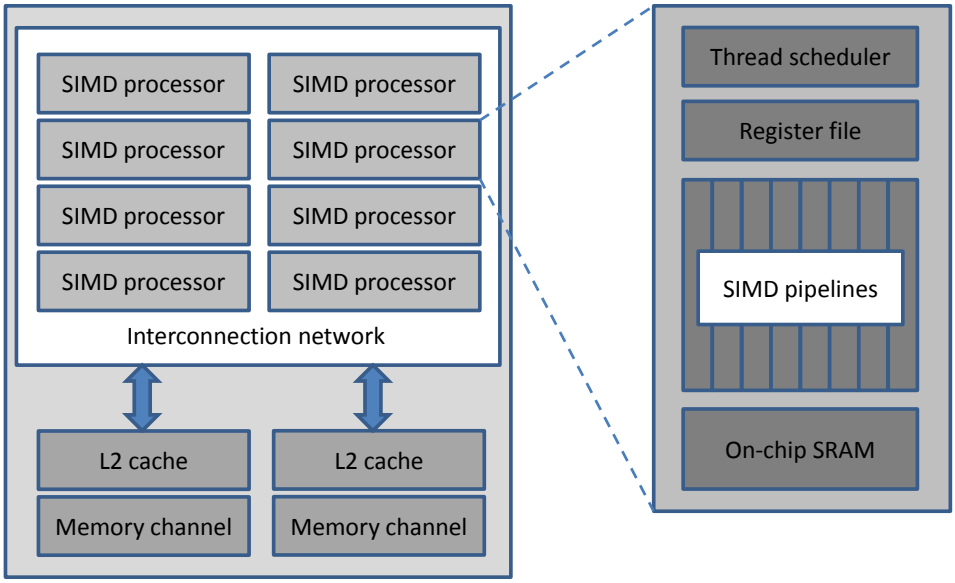


Figure 2.1: High-level block diagram of a modern GPU architecture

and other long latency operations with multi-threading. As a result, a large register file is needed to store the context of all resident threads. In NVIDIA GPUs, thread scheduling is done at the granularity of *warps* which is a group of 32 scalar threads. These warps have access to on-chip SRAM resources such as the L1 cache and the shared memory through multiple load store units. The shared memory is a software-controlled scratchpad memory that threads can use to communicate data.

2.2 GPU Programming and Execution Model

The most dominant GPU programming and execution model adopted by contemporary GPU architectures from major vendors like NVIDIA [5], AMD [6], and Intel [7] is commonly referred to as *single-instruction multiple-*

thread or SIMT. SIMT is similar to SIMD in that a single instruction is broadcast to multiple execution units to process multiple data elements in parallel. However in the SIMT programming model, each instruction stream mapping to a single SIMD lane is considered an independent thread which can follow its own control flow accessing arbitrary memory addresses. This provides an easier programming interface than SIMD by allowing the programmer to write thread-level parallel code for independent, scalar threads, as well as data-parallel code for coordinated threads [8].

With the SIMT programming model, a GPU kernel specifies the behavior of a single thread and is executed N times in parallel by N different scalar threads. Threads in a kernel form a hierarchy so that they map well to the underlying hardware described in Section 2.1. Figure 2.2 depicts the thread hierarchy defined in the Compute Unified Device Architecture (CUDA) programming model developed by NVIDIA [8].[1] A CUDA kernel is expressed as a *grid* of *thread blocks*. A SIMD processor executes multiple thread blocks concurrently provided that it has enough resources (e.g. register file and shared memory) to hold the context of all threads in those thread blocks. Threads within a thread block can cooperate by sharing data through the shared memory and by synchronizing their execution to coordinate memory accesses [8]. $\_\_syncthreads()$ is the CUDA intrinsic function for specifying synchronization points within the kernel. $\_\_syncthreads()$ acts as a barrier at which all

[1] Throughout this proposal, we assume an NVIDIA GPU and the CUDA programming model; however, the principals of our approach also apply to other GPUs and the OpenCL programming model [9].

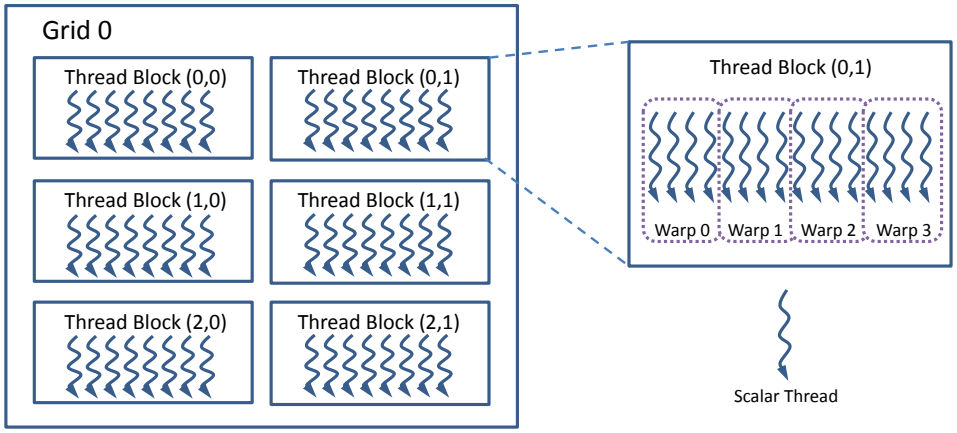


Figure 2.2: CUDA thread hierarchy

threads within the thread block must wait before any is allowed to proceed. A common usage is to read or write data into an array in the shared memory and to call *__syncthreads*() to ensure all reads/writes are seen by all threads within the same thread block. Although thread blocks are the finest grouping of threads exposed to the programmer, in the actual hardware, each thread block is decomposed into individual units of scheduling called *warps*. Each warp consists of 32 scalar threads in the current generation NVIDIA GPUs, and each SIMD processor selects an instruction from one or more warps each cycle and schedules them on the SIMD lanes.

2.2.1 Stack-Based Control Divergence Handling

As previously discussed, SIMT is different from SIMD in that threads within a warp can branch and execute independently of one another. When such a control divergence occurs, the warp serially executes each branch path taken, disabling threads that are not on that path by associating an *active*

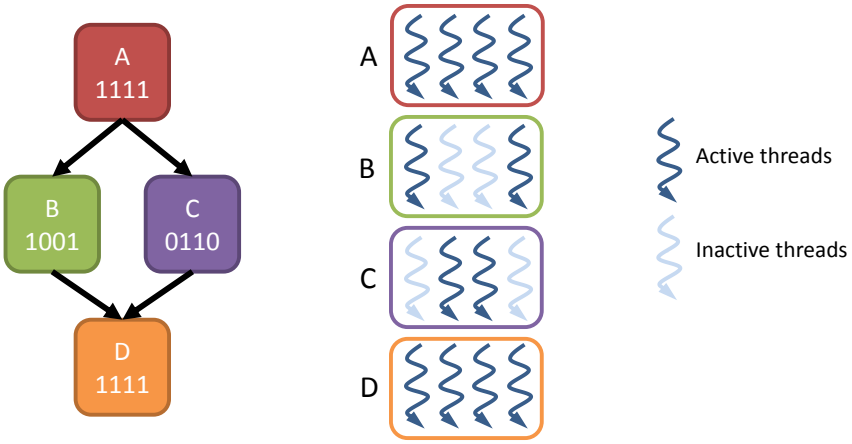


Figure 2.3: Control divergence within a warp causes serialized execution

mask with each path. Figure 2.3 shows an example control flow graph with potential divergence. If the branch condition at the end of the basic block A evaluates differently among threads within a warp, the warp serially executes each of the branch path B and C, converging at D. Inactive threads are masked out by the active mask so that they do not commit results.

A warp can potentially diverge to multiple paths and converge back arbitrarily at runtime. The management of divergent control paths is done by a hardware stack-based reconvergence mechanism in current NVIDIA GPUs [10]. First, reconvergence points are found by a static control flow analysis which identifies the *immediate post-dominator* instruction of each divergent branch. The immediate post-dominator instruction is the first instruction in the static control flow that is guaranteed to be on both diverged paths and thus can be seen as the reconvergence point [11]. (e.g. The immediate post-dominator of the branch instruction at the end of basic block A is the first instruction of

PC	Active mask	rePC
A	1111	-

(a) Initial stack status. All threads are executing basic block A.

PC	Active mask	rePC
D	1111	-
C	0110	D
B	1001	D

(b) At a divergent branch, both paths B and C are pushed on to the stack. Execution continues for the path on the top of the stack.

PC	Active mask	rePC
D	1111	-
C	0110	D

(c) The stack entry is popped when the PC value matches the rePC value. Execution continues for the next path.

PC	Active mask	rePC
D	1111	-

(d) All threads in the warp reconverge and basic block D.

Figure 2.4: SIMT stack-based reconvergence mechanism shown for the example control flow graph in Figure 2.3

basic block D in Figure 2.3.) This information is embedded into the instruction stream and passed to the hardware at runtime. In the hardware, execution paths of each warp are maintained in a stack structure, which we refer to as the SIMT stack, as shown in Figure 2.4. The SIMT stack stores (1) the program counter (PC) associated with each control path, (2) the active mask to indicate which threads are active, and (3) the reconvergence PC (rePC) to mark the recovergence point of each control path. Initially, all threads are in sync with a full active mask, and the rePC is set to the end of the kernel (Figure 2.4(a)). If the branch condition at the end of basic block A evaluates differently among threads within a warp, the PC field of the current top of the stack is updated to the rePC (D) of the branching instruction. Then active masks for each path B and C are calculated and pushed on to the stack along with the rePC value (Figure 2.4(b)). Once the stack manipulation is done, the execution path at the top of the stack (basic block B in this example) is chosen and run until the PC value matches the rePC value at which point the

stack entry is popped to continue execution of the next control path containing basic block C (Figure 2.4(c)). Similarly, when the PC value of execution path containing basic block C reaches D, the stack entry is popped and the control is *reconverged* at basic block D (Figure 2.4(d)).

2.3 The Problem of Design Margins (Guard-bands)

Traditional processors have pre-determined operating clock frequency and supply voltage pairs that are known to be safe. These operating points include sufficient design margin to guarantee correct operation for all input vectors under various noise processes such as PVT (process, voltage, and thermal) variations and aging effects. Design margins can be in the form of either higher supply voltage or lower clock frequency. Throughout this dissertation, margin will typically mean voltage margin and will be expressed as percent of nominal voltage value. Process variations are static and are caused by uncertainties in the semiconductor manufacturing process. An example of a process variation is random dopant fluctuation, which affects the threshold voltage of transistors leading to variations in circuit delay. Unlike process variation, voltage and thermal variations are dynamic and are affected by characteristics of the running application as well as the external environment. For example, sudden changes in switching activity can cause large voltage fluctuations or temperature changes within a processor. At a high level, all of these variations appear as changes in circuit delay and result in timing errors if the margin is too small.

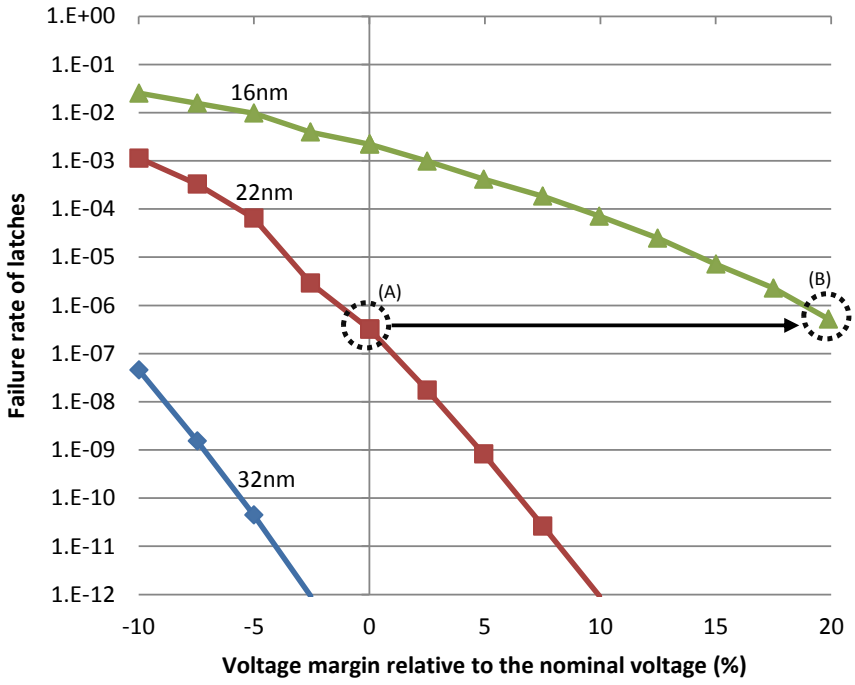


Figure 2.5: Latch failure rates due to process variation [12]

Since design margins are set based on worst-case scenarios, the processor runs at a sub-optimal operating point in terms of energy efficiency under typical conditions. Technology scaling exacerbates the problem because of increased process variation, decreased supply voltage, and higher peak current draw. Figure 2.5 shows the ITRS projections for power-supply dependent failure rates of latches due to process variation [12]. Note that a latch is considered faulty when the clock to output delay is $10X$ the nominal delay. At the $22nm$ technology node, one out of a few million latches is expected to fail assuming a 0% voltage margin as shown in Figure 2.5 with dotted circle (A). However,

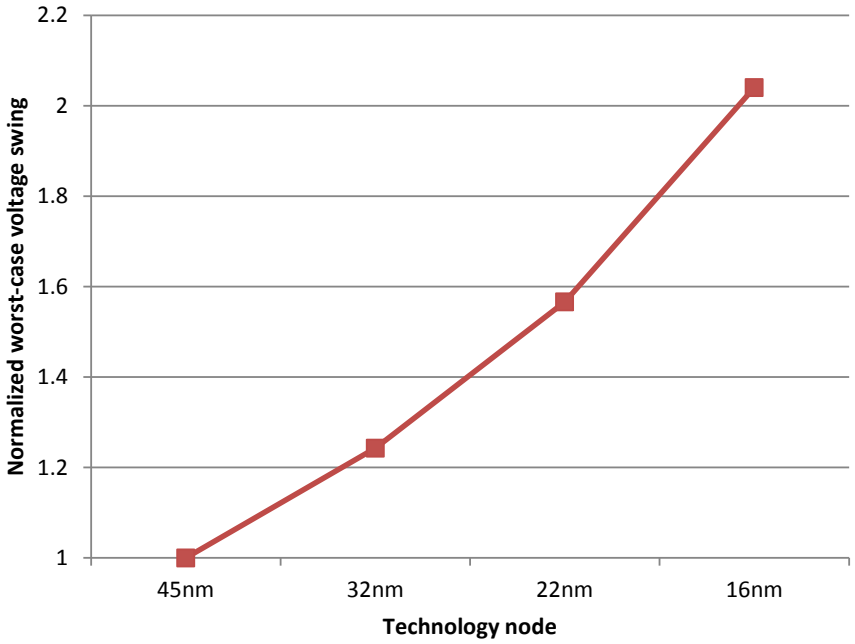


Figure 2.6: Worst-case peak-to-peak voltage swing [13]

a 20% voltage margin relative to the nominal voltage is required to maintain the same failure rate at the 16nm node, as illustrated by dotted circle (B).

Technology scaling also increases voltage variation thereby requiring larger design margins. Voltage droop due to on-chip inductance, also known as di/dt noise already takes up a significant portion of the design margin (20%) in modern microprocessors [14]. This fraction is expected to grow as the supply voltage scales and the peak current draw increases. Simulations based on the ITRS projections [13], as shown in Figure 2.6, indicate that the worst case voltage swing will rapidly grow for a modern power delivery package design. These predictions imply that the traditional worst-case designs may not scale

beyond the next few generations and that a new design paradigm is needed to keep the benefits of technology scaling.

2.4 Better Than Worst-Case Design

A new class of design strategy, often called better-than-worst-case design, [15] has recently emerged as a promising alternative to the traditional worst-case design methodologies. Better-than-worst-case design tries to solve the problem of design margins by optimizing the system for the typical-case instead of for the worst-case. For typical scenarios, the circuit can run with a reduced margin without generating any errors, leading to better energy efficiency. In order to guarantee correctness in atypical cases, errors are either avoided with prediction or corrected with detection and recovery. In this section, we refer to the former as avoidance approach and the latter as reactive approach and classify previous work on better than worst-case design into these two categories.

2.4.1 Avoidance Approaches

Avoidance approaches use error predictors to predict errors before they occur and take preventive actions such as throttling. Examples of error predictors include processor thermal sensors [16, 17], voltage sensors [18], current sensors [19, 20], and critical path monitors (CPMs) [21, 22] or tunable replica circuits (TRCs) [23]. CPMs or TRCs are synthetic circuit elements consisting of a number of logic gates and wires that can be tuned to mimic the criti-

cal path of the circuit. IBM Power7 is an example system that employs the avoidance approach with CPMs [22]. The CPMs in Power7 measure available timing margins at runtime and notify the clock and voltage controller so that the system can dynamically adjust to a more efficient operating point. A recent work by Reddi et al. [24] proposes a signature-based error predictor that uses microarchitectural events and program control flow to generate signatures of voltage emergencies and throttle the processor to prevent errors from occurring. Although avoidance based, the predictors in this approach are not perfect, and a voltage sensor and a reactive checkpoint-recovery mechanism is required to recover from occasional mispredictions as well as to build the initial voltage emergency signatures.

Avoidance approaches generally work well for reducing margins for static process variations since hardware-based error predictors can be tuned post-fabrication. Furthermore, margins for slow-changing error processes such as thermal emergencies and aging effects can also be effectively reduced based on prediction since it is usually possible to adjust the clock or voltage before errors occur. However, voltage variations at runtime can be a much quicker event and thus it is more difficult to guarantee complete error avoidance where adjustment of clock and voltage can take many cycles.

2.4.2 Reactive Approaches (Timing Speculation)

Reactive approaches use error detectors to detect already-occurred errors and rely on recovery mechanisms to maintain correctness [3, 25, 26].

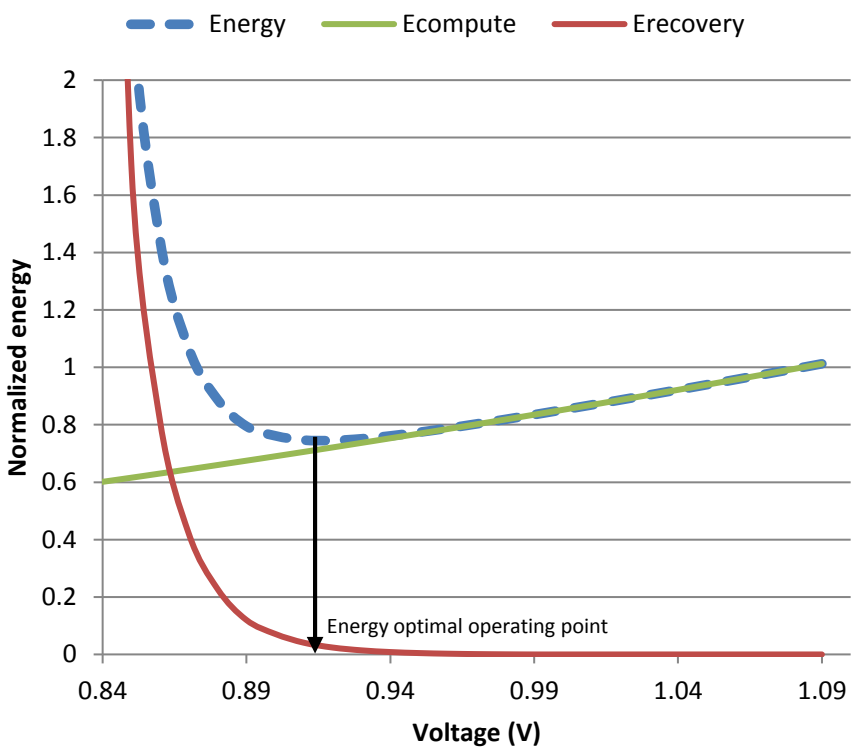


Figure 2.7: Energy efficiency optimized with timing speculation

Due to their ability to recover, they can be applied to fast-changing error processes such as voltage emergencies further reducing the margins that avoidance approaches cannot easily address. In general, reactive approaches are a *timing speculation* technique in that the circuit timing is speculated to be met and misspeculations are handled with specially designed detection and recovery mechanisms. The efficiency of timing speculation is determined by the amount of voltage reduction, the error rate with respect to the voltage, and the extra runtime overhead caused by recovery. Careful trade-off analysis must be performed to derive the optimal operating point (e.g. voltage) for a

given workload. An illustrative example of efficiency optimization is shown in Figure 2.7. The energy required for computation, $E_{compute}$, decreases quadratically as the operating voltage is reduced. However, reducing voltage also leads to errors at runtime increasing the energy required for recovery, $E_{recovery}$. Sum of $E_{compute}$ and $E_{recovery}$ gives the energy consumption of the running workload, and the energy optimal operating point is the voltage that minimizes the energy consumption.

A number of timing speculation schemes have been proposed since it was first introduced by Ernst et al. [25]. At a high-level, timing speculation schemes are a combination of error detection and error recovery mechanisms which are often closely related to each other. The following sections describe previously proposed timing speculation schemes and briefly discuss how fine-grained containment domains overcome shortcomings of prior approaches.

2.4.2.1 Razor

Razor [25], proposed by Ernst et al., adds a shadow latch driven by a delayed clock to each flip-flop in the timing critical path of the circuit. Timing error is detected when the output of the shadow latch does not match the output the flip-flop. The shadow latch is designed to latch the correct result using a delayed clock; therefore, a single-cycle stall lets the erroneous path recompute its result using the correct result at the output of the shadow latch. Applying the same approach to SIMD pipelines has limited advantage because the cost of recovery is multiplied by the SIMD width. Krimer et al. discussed

the implications of timing speculation on wide-SIMD architectures and proposed to decouple SIMD lanes to allow independent recovery of each lane in case of timing errors [27]. While shadow latch based detection allows simple stall based recovery with minimum runtime overhead, there are implementation costs associated with it. Due to the design of the shadow latch based detector, there is a possibility that the flip-flop could become meta-stable and thus extra circuitry is required to detect meta-stability of the flip-flop. Another shortcoming is that the stall signal propagation delay could limit the clock frequency of the design, especially for complex high performance processor with deep pipelines. Lastly and most importantly, shadow latch based detectors introduce additional minimum delay (hold time) timing constraints [27]. If a path in the combinational logic is too short, the computed value can reach the shadow latch before the delayed-clock generating a false alarm. In order to solve this problem, buffers must be inserted to short paths adding area and power overheads.

2.4.2.2 Razor Variants with Architecture Recovery

Different types of error detecting flip-flops have been designed [26, 28] to overcome the shortcomings of the original Razor [25]. These flip-flops eliminate the meta-stability problem but lose the ability to recover with a single cycle pipeline stall. Instead, timing speculation schemes utilizing these detectors employ an architecture-level instruction replay mechanism to recover from errors. When an error is detected, the pipeline is flushed and the program

counter (PC) is updated to the PC of the erroneous instruction for replay. Xin et al. [29] observe that error rates are different for different static instructions and build a predictor to identify timing critical instructions and selectively stall those instructions, thereby reducing the cost of architecture-level instruction replay.

In an out-of-order processor, the built-in support for speculative execution can be leveraged to recover from errors in a similar way that it recovers from branch mispredictions [30, 31]. Such a mechanism, however, is not present on GPUs and implementing it is very expensive due to the large amount of thread context that needs to be preserved.

2.4.2.3 DeCoR

DeCoR [3], proposed by Gupta et al., is a delayed commit and rollback recovery mechanism for handling voltage emergencies in processors. In their approach, voltage sensors are used to detect voltage emergencies at runtime and a delayed commit architecture is designed to prevent errors from writing to memory or architectural state. When a voltage emergency occurs, speculative states are flushed and the program rolls back to the last noise-verified state in a similar way that branch mispredictions are handled in modern out-of-order processors. Similarly to the techniques discussed in Section 2.4.2.2, DeCoR relies on speculation hardware to implement efficient recovery which is not present on GPUs.

2.5 Overview of Resilience Schemes

Although not directly applicable to the problem of design margins, there is a broader range of literature on resilience that is related to this dissertation. In this section, we discuss various resilience schemes in relation to the types of errors they cover, and the detection and the recovery mechanisms they employ.

2.5.1 Errors and Failure Mechanisms

Errors that affect semiconductor devices can be broadly classified into three categories: hard errors, soft errors, and intermittent errors. The failure mechanisms for hard errors are permanent stuck-at faults that occur in the field, undetected manufacturing or design flaws, or degradation-dependent faults that initially look like transient errors but become permanent under further degradation. This type of error causes permanent removal of a component and may trigger reconfiguration of the system.

The failure mechanisms for soft errors, also known as transient errors, are energetic particle strikes on a sensitive node in a micro-electric device which cause hole-electron pairs to be generated, effectively injecting a momentary ($< 1ns$) pulse of current into a circuit node. This results in a single event upset (SEU) where a bit stored in a storage cell (e.g. memory or flip-flop) flips, or the combinational logic writes a wrong value into the latch or flip-flop.

The failure mechanisms for intermittent errors are, as discussed in Section 2.3, variations introduced during manufacture (e.g. process variations) and runtime (voltage and thermal variations) that cause temporal timing violations along the critical paths of the logic. Intermittent errors are becoming more serious as the process technology scales, and also as we push the margin with techniques like dynamic voltage and frequency scaling (DVFS) [32]. This dissertation focuses on designing a system that can handle intermittent errors in an efficient manner thereby addressing the problem of design margins.

2.5.2 Checkpoint-Recovery

Checkpoint-recovery is a generic state preservation and restoration mechanism, and is often employed to provide error recovery in large-scale compute clusters. Traditional coarse-grained checkpointing takes a snapshot of the application state and stores it to persistent storage such as a disk. System-level checkpointing [33, 34, 35, 36] relies on the operating system or runtime to checkpoint the raw bytes of the application memory whereas application-level checkpointing [37, 38] modifies the application source code to insert checkpoint and recovery routines. To automate this instrumentation process, compiler-based approaches have also been proposed to make an application self-checkpointing and self-restarting [39, 40]. Orthogonal to the application-intrusiveness, checkpoints can be global or local. Global checkpointing establishes a synchronized program state of every node in a centralized storage like the global file system. To overcome the inefficiencies associated with central-

ized global checkpoints, researchers have proposed local checkpointing. Local checkpointing stores the state of each node locally in non-volatile storage. While faster and more scalable than global checkpointing, a naive implementation cannot recover from permanent node failures. Hence global checkpointing is often combined with local checkpointing to tolerate such errors. Generic reliability models for two-level local/global checkpointing are studied in [41, 42].

Checkpointing in a distributed environment requires coordination between processes to create a consistent system state which is generally accomplished with global synchronization. Same principles apply to shared memory multiprocessors where a complete copy of the system's architectural state is periodically checkpointed [43, 44] in a synchronized manner. To eliminate the need for global synchronization, researchers have proposed uncoordinated checkpointing where each process maintains multiple checkpoints independently, delegating construction of consistent system state to recovery. This approach may result in the *domino effect* [45]; i.e., a single error local to one process can cause all processes to use up all of their checkpoints, and various approaches have been proposed to eliminate the domino effect [46, 47, 48, 49].

Due to high overhead of taking a checkpoint, these coarse-grained system-level checkpointing schemes described above are only effective for recovering from relatively infrequent errors such as power failures or radiation-induced soft errors. To deal with other types of errors, checkpointing can also be done at a finer granularity for individual processors. Micro-architectural checkpointing saves the architectural state in a hardware buffer to enable fast

recovery in case of an event like branch misprediction [30, 31]. Similar mechanisms can be used to enable timing speculation as discussed in Section 2.4.2, to recover from soft errors [50], or to support online diagnostics of hard faults [2]. Micro-architectural checkpointing usually relies on speculative hardware structures such as the reorder buffer (ROB) and the load store queue (LSQ) that are only present in out-of-order processors and thus not suitable for throughput processors such as GPUs.

FGCDs can also be considered a checkpoint-recovery mechanism in that preservation and restoration of system state is used to recover from errors. At a high-level, FGCDs are an application-level, local, and uncoordinated checkpointing scheme that work at a granularity finer than the system-level checkpointing and coarser than the micro-architectural checkpointing.

2.5.2.1 Transactional Memory

Recent research on transactional memory (TM) [51, 52, 53] mainly aims to enable efficient concurrent and lock-free programming. Transactional memories can be regarded as a checkpoint-recovery mechanism since state preservation and restoration are inherently provided at transaction boundaries. Relax [54] and FaulTM [55] extend transactional memory concepts to resilience by taking a cooperative hardware-software approach. Relax uses *try/catch* like semantics to define transactions. The programmer can *relax* a block of code and define recovery behavior for the block while error detection is assumed to be performed by a low-latency hardware error detector. Similarly, FaulTM lets

the programmer declare a vulnerable block and rely on hardware transactional memory for preservation and restoration of the vulnerable block. Neither approaches however, exploits nested hierarchy for localizing failures to the closest domain, nor do they allow for further application or machine-specific optimizations which are enabled by FGCDs.

2.5.2.2 Idempotence-based Recovery

Implementing architecture recovery on in-order processors and GPUs is potentially challenging because the techniques available for out-of-order processors are very expensive to implement on simple pipelines. Recent proposals on in-order processors [56] and GPUs [57] revisit the concept of idempotence to implement software-based fault recovery on processors without hardware speculation support. Idempotence is the property that a code region can be executed multiple times without side effects thereby producing the same final result. As formally defined by De Kruijf et al., a code region is idempotent if there are no *clobber antidependences*, where a clobber antidependence is defined as a WAR (write-after-read) dependence without prior RAW (read-after-write) dependences on the same variable [58]. Figure 2.8 shows examples of idempotent and non-idempotent regions of code. Figure 2.8(a) is idempotent because $R0$ is *protected* by a RAW dependence before the WAR dependence introduced by the last statement. Re-execution of this code region will always start by initializing $R0$ to 1 and will generate the same result. However, Figure 2.8(b) lacks the initial write to $R0$ hence a clobber antidependence exists

```
R0 = 1;
R1 = R0 + R0;
R0 = R2;
```
(a) Idempotent

```
R1 = R0 + R0;
R0 = R2;
```
(b) Non-idempotent

Figure 2.8: Idempotence of code regions based on the dataflow

on variable $R0$. In order to make this region idempotent, the value of $R0$ must be preserved at the beginning of the region. Idempotence-based recovery approaches decompose a program into a series of code regions and add preservation instructions to make every code region idempotent.

It is interesting to note that many applications are composed of idempotent code regions with relatively small amount of live-in state, and recovery can be done at the granularity of idempotent regions without the overhead of checkpointing the entire context. Encore [56], presents a fully software based transient fault recovery using compiler level idempotence analysis and applies it to simple in-order processors. A similar approach has been proposed for GPUs to support exceptions as well as speculative execution [57]. Although idempotence analysis is a very useful tool in finding a reasonable point of recovery with minimum preservation overhead, purely relying on compiler analysis limits the flexibility and can be suboptimal in the context of timing speculation. More specifically, idempotence analysis does not have the notion of nesting making it difficult to take advantage of the machine and application hierarchy. For example, the frequency of preservation within a loop cannot be tuned to achieve the best trade-off between preservation and recovery over-

head. It is also difficult to support arbitrary error detection latencies because the processor must stall at the boundaries of idempotent regions until all instructions in that region are verified to be correct by the error detector.

This dissertation shares the same observation that only a small amount of state needs to be preserved for recovery. However, instead of relying on compile-time idempotence analysis, I propose a programming construct to let the programmer specify the hierarchy of code regions mapping to CDs. This way, the preservation interval can be tuned to achieve the best efficiency for a given error rate giving the flexibility to adapt to a wide range of environmental conditions and operating scenarios. The hierarchy of CDs also acts as a safety net such that any missed errors (e.g. in case of long latency error detectors [59]) in a CD can still be recovered by re-executing its *parent* CD. Another benefit of having a programming construct is that high-level knowledge available at the source code level can be exploited to enable more efficient resilience mechanisms such as elimination of redundant preservation data and adoption of low-cost, application-specific error detection mechanisms.

2.5.3 Pipeline-Level Recovery

Pipeline stall is a simple technique to avoid timing errors. When a timing error is detected, the pipeline is stalled for an additional cycle to allow enough time for the correct value to be propagated to the next pipeline stage. While this method effectively implements a single cycle recovery, a global stall

signal must be computed and transmitted to all pipeline stages to guarantee correct recovery. Razor [25] uses pipeline stalls to recover from errors.

Counterflow pipelining is another pipeline-level recovery technique already present in many modern processor pipelines to support instruction flush and replay [60]. Error recovery works in a similar manner. When an error is detected at a pipeline stage, all instructions in the preceding pipeline stages are flushed by backwards propagating a flush signal and overwriting the program counter (PC) register with the PC of the erroneous instruction. While counterflow pipelining does not require a global stall signal to be propagated, the overhead of flush is higher than stall.

Pipeline-level recoveries require error detection to be done immediately at the granularity of pipeline stages and thus they are usually paired with error detecting flip-flops used in [25, 26, 28]

2.5.4 Algorithm Based Fault Tolerance (ABFT)

Algorithmic-based checking allows for cost-effective fault tolerance by embedding a tailored checking, and possibly correcting, scheme within the algorithm to be performed. It relies on a modified form of the algorithm that operates on redundantly encoded data, and that can decode the results to check for errors which might have occurred during execution. Since the redundancy coding is tailored to a specific algorithm, various trade-offs between accuracy and cost can be made by the user [61, 62]. Therein also lies this techniques main weakness as it is not applicable to arbitrary programs

and requires time-consuming algorithm development. In the case of linear algorithms amenable to compiler analysis, an automatic technique for ABFT synthesis was introduced in [61].

2.5.5 Approximate Computing

Previously discussed resilience schemes are designed to guarantee fully precise computation results through error detection and recovery. Approximate computing however, improves energy efficiency by introducing imprecision tolerance into the application. Approximate computing is related to resilience because it operates at a regime of deliberate extreme low reliability; therefore, some techniques for approximate computing may also be used to achieve correct forward progress when unintended errors occur [63].

Approximate computing applications such as multimedia processing and machine learning can naturally tolerate errors as long as the quality constraints given by the application are met, and as a result, substantial efficiency gains are possible. Approximate computing can be applied at different levels. At the hardware level, approximate adders [64] and multipliers [65, 66] with reduced precision can be used for instructions that do not require full precision. At the architecture level, a set of approximate instructions can be provided to enable collaboration between the language, compiler, and architecture [67]. At the algorithm level, certain computations can be skipped still generating an acceptable result. This is especially true for many iterative algorithms that incrementally refine the result [68, 69]. Recently, a programming language

that enables developers to specify reliability requirements of a function has also been proposed [70] which can be used to quantitatively verify a program's reliability running on unreliable hardware.

Chapter 3

System Assumptions

To set the context in which the FGCDs are applied and evaluated, this chapter describes the basic system assumptions that we make. This includes the type of errors that FGCDs target and assumptions on their properties such as the failure and manifest mechanism, the error rate model, the distribution of errors, and the error detection mechanism. We also explain our assumptions about system-level control mechanisms, such as voltage control, as well as mechanisms for preventing recurring errors.

3.1 Error Properties

FGCDs can be used to tolerate any kind of errors as long as the system can provide error containment with respect to the CD hierarchy. However, as discussed in Chapter 1, this dissertation aims to solve the problem of design margins by enabling timing speculation with FGCDs. Thus, we focus on timing errors that are caused due to insufficient design margins. Specifically, FGCDs target voltage noise margins that are required to protect circuits against di/dt noise, or voltage droops. We assume that memory cells are protected with Error Checking and Correcting (ECC) codes and voltage droops

only manifest as timing errors along the critical paths of the combinational logic. At the architecture level, errors result in wrong values being written to the architectural state leading to silent data corruption (SDC) or control flow errors that cause the system to crash.

3.1.1 Error Rate Model

As explained in Section 2.4.2, the efficiency of timing speculation is largely affected by the error model that defines the relationship between supply voltage and error rate. Reduction of voltage results in increased circuit delay, and leads to timing error if a circuit path fails to meet the timing constraint due to the increased delay. The error rate at a given voltage depends on how often such a circuit path is exercised at runtime. Previous studies show that there is an exponential relationship between supply voltage and error rate. Krimer et al. [27] built an exponential error probability model and validated the model with various adder and multiplier designs. Measurements performed on microprocessors by Das et al. [26] and Bacha et al. [71] also confirms the exponential relationship between voltage and error rate.

In our evaluation, we derive an error model by fitting the data presented in [71] to an exponential curve as shown in Figure 3.1. The data is gathered on an 8-core Intel Itanium processor fabricated in 32nm process technology with a nominal voltage of 1.1V. The unit of error rate represents the expected number of errors per core per cycle and we apply this model to each SIMD processor in our GPU design. According to our simulation results, FGCDs usually achieve

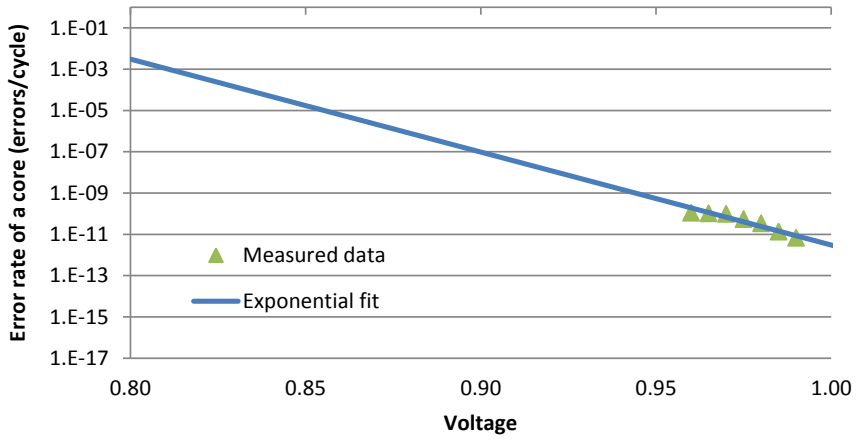


Figure 3.1: Error model

the highest efficiency at an error rate of 10^{-5} errors/cycle or below for most applications. FGCDs are effective for a range of error rates that are higher than soft error rates or power failure rates that the system-level checkpoint-recovery mechanisms target ($< 10^{-9}$ errors/cycle) [1], and lower than error rates that pipeline recovery based timing speculation techniques like Razor targets ($> 10^{-2}$ errors/cycle) [25], which relates to input vector dependent delay distributions of the circuit.

As the measured data in Figure 3.1 shows, the processor does not experience any error until the voltage reaches 0.99V, 10% below the nominal voltage. We assume that this 10% margin is for static process variations and slower temperature variations. As discussed in Section 2.4, this margin can be effectively reduced using an avoidance approach that employs calibrated sensors as error predictors. Thus, in our evaluation, we use 1.0V as our base-

line operating point and normalize performance and energy efficiencies of my approach to the baseline.

Note that the error model in Figure 3.1 is used as a reasonable reference to show the applicability of FGCDs to timing speculation. In reality, the error model can vary depending on various factors such as the processor design, the package design, the fabrication technology, and the dynamic temperature that the processor in running at. Sensitivity of FGCDs to different error models is left to be studied as a future work.

3.1.2 Error Distribution

In our simulation-based experiments, errors are injected to each SIMD processor with an error probability corresponding to the voltage that the processor is operating at, as shown in Figure 3.1. Given an error rate, each benchmark is run for a sufficient number of cycles to experience at least a few errors during the course of simulation. For the purpose of error injection, we assume errors have an uniform random distribution and SIMD processors are independent of one another. Thus, in our evaluation, all instructions are equally likely to experience errors, and error rate of a CD is solely dependent upon the number of instructions in the CD.

Previous studies [72, 73, 24, 74, 13] suggest that voltage droops are not random events, but are related to the characteristics of the running workload which allows an prediction mechanism to be developed. Furthermore, different function units in hardware may have different susceptibility to timing errors

depending on their design [27]. FGCDs can exploit these properties to further enhance the efficiency by adjusting the error rates of each CD based on its predicted susceptibility and tuning the application based on the adjusted error rates.

3.1.3 Error Detection

Timing speculation techniques based on pipeline-level recovery require error detecting sequentials such as the Razor flip-flip to isolate timing errors in pipeline stages. Since these detectors have to be placed in all critical paths of the logic, they are intrusive to the design and can have high overhead. Furthermore, incorporating pipeline-recovery in a throughput architecture may require modifications to the SIMD semantics as discussed by Krimer et al. [27].

Instead, we use hardware sensor based error detection that has lower overhead and is not intrusive to the design. Previous works on avoiding voltage emergencies suggest error detectors based on current sensing [20, 19] or voltage sensing [18]. These detectors can be designed to trade-off the accuracy against the latency. For example, the sensor design discussed in [20] computes the voltage by performing a convolution of power supply current over 350 cycles, which is the length of the impulse response that is being modeled. The authors show reduced, but acceptable, accuracies when only the first 25 or 43 elements are taken into convolution. For faster detection, Joseph et al. [18] also proposed alternative detection circuits based on buffer delay lines or inverter chains,

similar to the CPMs or TRCs explained in Section 2.4.1 which can provide 1-2 cycles latency detection.

In this dissertation, we assume generic error detectors with adjustable detection latency are available at the granularity of SIMD processors. We further assume that error detection is always precise, and the latency can range from zero to fifty processor cycles to show sensitivity of the proposed scheme against widely varying detection latency.

3.2 System Control Mechanisms

In this section, we discuss various system-level control mechanisms that are essential in a timing-speculation environment.

3.2.1 Voltage Control

The GPU architecture we assume uses a single power delivery network that is shared by all SIMD processors, based on the observations made by Leng et al. [75]. Thus, the voltage control is done at the chip level, and since chip-level voltage regulator modules are usually slow (voltage change occurs in the order of microseconds [76]), FGCDs find a single optimal voltage for an entire GPU kernel. Recent advances in on-chip voltage regulator modules may enable much faster voltage regulation [76] allowing intra-kernel voltage adjustment. The evaluation and optimization of such approaches are left for future work.

Also note that voltages are adjusted in 10mV steps as in [75]. Our optimization results shown in the later section (Section 6.1) show that 10mV is fine-grained enough to capture the optimal operating points.

3.2.2 Handling Recurring Errors

Voltage droops are related to the characteristics of the running workload, and simply replaying the same execution for error recovery may result in recurring errors. We expect this effect to be less significant for GPUs since a large number of threads are being interleaved, naturally leading to a different supply current profile on a recovery. To further ensure that recurring errors are avoided, we assume throttling of the processor for a short period of time on a recovery. Previous work [3] shows that running the processor for 10 cycles at the 50% frequency is enough to guarantee forward progress.

Chapter 4

Fine-Grained Containment Domains for Throughput Processors[1]

Fine-grained containment domains are a hardware/software cooperative approach to efficiently implement hierarchical state preservation and restoration as well as error recovery on GPUs. With FGCDs, GPUs can tolerate high and widely varying error rates with predictable performance overheads, and in consequence, it is possible to adopt the idea of timing speculation for better energy efficiency. This chapter discusses the semantics of FGCDs, mapping of an application to CDs, and tuning of a given mapping for optimal efficiency using an analytical model. Then, the design and the implementation of software and hardware components organizing the FGCDs are given.

4.1 Semantics of FGCDs

Fine-grained containment domains work by decomposing a program into a nested hierarchy of CDs as illustrated by Figure 4.1. Each CD consists

[1]The general concept of containment domains discussed in this chapter is based on the prior publication [1]. The general concept and semantics are developed through open discussions among all authors. Jinsuk Chung developed the analytical model, Ikhwan Lee defined the system parameters and error/fault model, and Michael Sullivan, Jee Ho Ryoo, and Dong Wan Kim mapped applications to containment domains.

of three explicit components: *preserve*, *compute*, and *recover*. The *preserve* component locally and selectively preserves state required for recovery. The preservation storage space within the GPU memory hierarchy is configured via an API call based on the capacity and the bandwidth requirement. The *compute* component, which contains the actual work to be performed by the program, is then executed. The *recover* component is initiated when an error is detected.[2] The initiation of recovery is done by a hardware mechanism that directs the control of affected threads to the beginning of the *recover* component. The *recover* component restores the preserved state and jumps to the *compute* body to perform re-execution.

As explained in Section 1, CDs have weak transactional semantics and are designed to be hierarchically nested; failures in an inner domain are encapsulated and recovered by that inner domain whenever efficiently possible. Erroneous data is conceptually never communicated outside of a CD, and there is no risk of an error escaping containment. Because of constraints on the error detection latency and the need for inter-CD communication, some CDs can be too costly to recover at a fine granularity. For such cases, an inner CD *escalates* the error to its parent, which in turn may escalate it further up the hierarchy until some CD can recover the error efficiently. For example, a CD must wait at the end of the domain until all data generated in the domain is verified to be correct, and this may be too costly for short CDs if the error detection

[2]In this dissertation, we assume a concurrent hardware error detector that can detect and report errors at any point in the program execution with a certain detection latency.

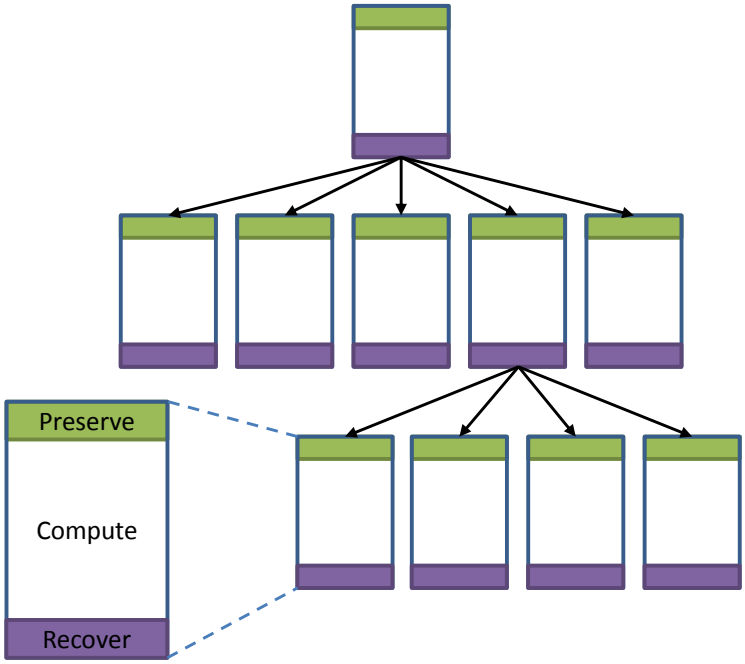


Figure 4.1: The organization of hierarchical CDs. Each CD has three components. The relative time spent in each component is not to scale.

latency is long. Instead of waiting, the inner CD can speculatively proceed to the next CD and escalate recovery to the parent if errors are later detected and reported to be associated with the previous CD. Detailed discussion of escalation is given in Section 4.4.2.3. Below we restate the core semantics of containment domains:

1. Any error occurring while a CD is running must be detected and associated with the CD regardless of the detection latency.

2. Any detected error triggers recovery by either:

   (a) using the recover component of the CD that detected the error, or

(b) escalating to the parent CD for recursive recovery.

3. Communication between threads must obey the following rules:

   (a) Threads belonging to the same CD may communicate freely.

   (b) Threads belonging to different CDs may not communicate unless such communication is inconsequential to the application after error recovery; e.g., multiple CDs atomically updating a minimum value at a shared memory location can be freely re-executed as long as the updates are error-free.

   (c) Inter-CD communication must be free of errors.

4. A CD completes once all potential detectable errors and failures within it have been detected. Upon completion, all preserved state is freed.

As a result of these semantics, FGCDs offer hierarchical and uncoordinated local checkpoint-recovery, which is an important distinction from prior work on transactional memory [54, 55] or compiler-based idempotence analysis [56, 57] where there is no notion of hierarchy. This allows for an extra tuning opportunity which makes FGCDs more efficient and flexible.

4.1.1 Recovery Scope

A CD may contain multiple threads potentially communicating with one another. On error recovery, all threads belonging to the CD must roll back together to maintain consistency. It is thus desirable to map only threads that

must communicate one another to the same CD. We use the terms that represent the thread hierarchy in the CUDA programming model, namely *warp*, *thread_block*, and *grid*, to describe the communication scope and thus the *recovery scope* of a CD. The recovery scope can be seen as the collection of threads, or the amount of parallel work, that needs to rollback together in case of an error. Note that this is different from the length of the CD, which represents the amount of serial work required for recovery. When there is no communication outside of a warp, errors are contained within the warp and the erroneous warp can recover independently in an uncoordinated fashion. If inter-warp communication exists in a CD, such as the use of shared memory writes followed by a thread block wide barrier *__syncthreads*() call as shown in Figure 4.2, errors in a warp might have affected other warps in the thread block requiring a thread-block-scope recovery. Similarly for global communication, the grid-wide recovery scope is assigned. Technically, grid-scope recovery will always result in re-execution of the entire kernel due to the programming model of GPUs that does not allow global synchronization within a kernel. However, grid-scope recovery is still useful when a kernel launches multiple kernels such as in the case of CUDA dynamic parallelism [77].

As described in Section 2.2, the GPU programming model is hierarchical such that it makes it easy to reason about the mapping of a parallel workload on to the underlying hierarchical hardware. As a result, in many GPU applications, communication patterns between thread groups exhibit locality and are clearly expressed with explicit synchronization or communi-

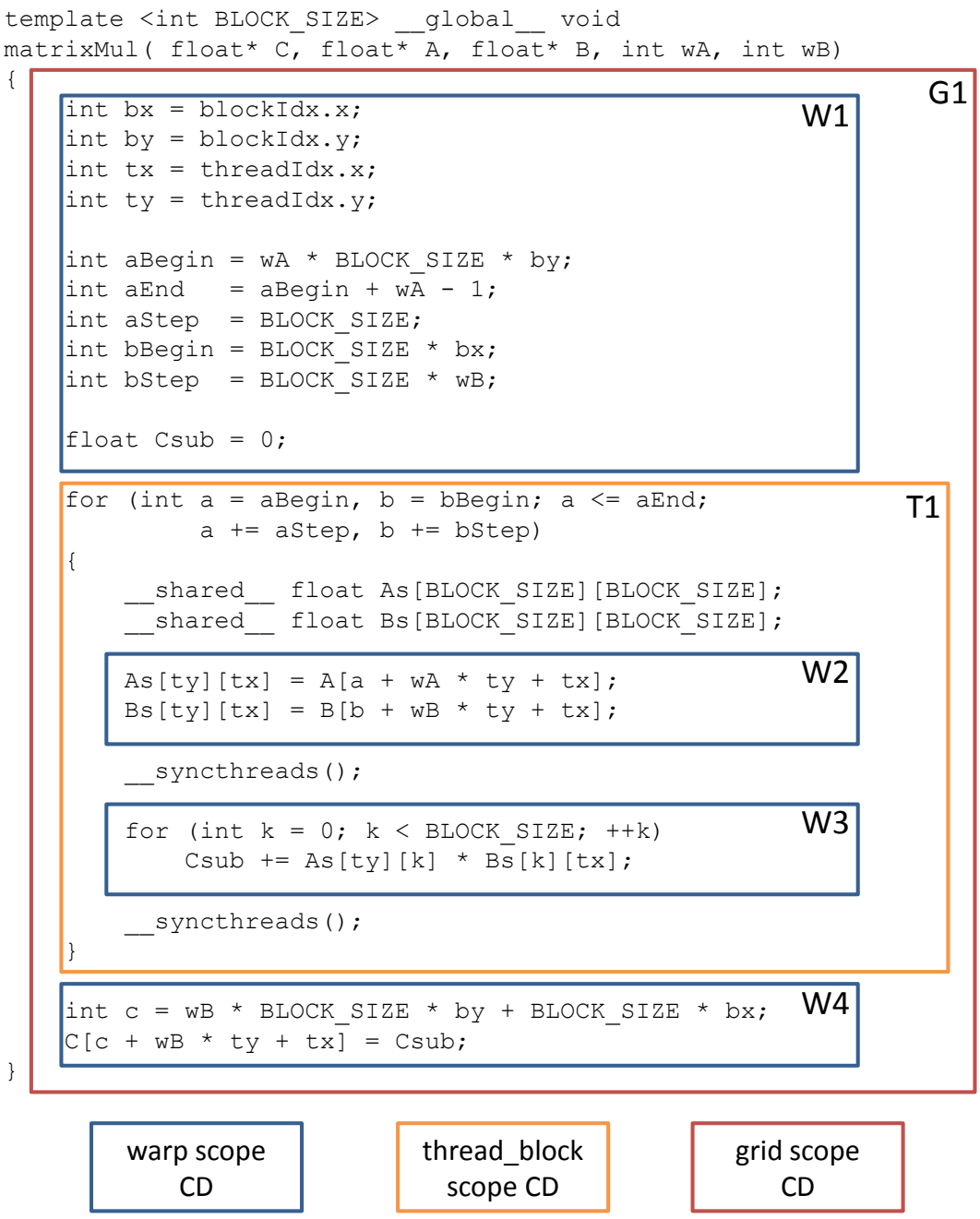


Figure 4.2: Example CD mapping of the matrix multiplication kernel in the CUDA SDK [78].

cation primitives, allowing FGCDs to efficiently limit the scope of recovery. Figure 4.2 shows an example hierarchical mapping of CDs with different recovery scopes for the *matrix multiplication* kernel included in the NVIDIA CUDA SDK (Software Development Kit) [78]. The kernel is mapped to hierarchical CDs G1, T1, W1, W2, W3, and W4. The main computation of the kernel happens in W2 and W3. W2 reads a sub-block from each of the input matrices A and B and stores them in the shared memory arrays As and Bs. Then, the sub-blocks are multiplied to calculate the partial sum Csub in W3. The first *__syncthreads*() call ensures that all sub-blocks have been loaded before consuming them, and the second *__syncthreads*() call guarantees that all sub-block values have been consumed before being overwritten in the next iteration. Consequently, T1 needs to have a thread block scope recovery to handle escalation from W2 or W3. Note that G1 is set to have a grid-scope recovery for illustration purpose only. In reality, all thread blocks in the matrix multiplication kernel execute independently, and thus thread-block-scope recovery is enough for the outermost CD.

Since threads within a CD must recover in a coordinated manner, threads may not proceed to the next CD until all other threads in the same CD have also reached the end of the CD, free of errors. This restriction imposes a non-programmatic barrier at the end of each CD. This is not a problem for warp-scope CDs because threads within a warp always execute synchronously on GPU hardware. A special case of divergent threads within a warp is handled by a hardware mechanism and will be explained in Section 4.4.1. Grid-scope

CDs are also not a problem because by definition they are always placed at the kernel boundary. For CDs with thread-block-scope recovery, this extra synchronization might have an impact on the performance. This overhead can be minimized by carefully placing CD boundaries where programmatic barriers already exist, as shown in Figure 4.2. Our evaluation shows that, for the applications studied in this dissertation, this is in fact a natural CD mapping that leads to optimal efficiency.

4.2 CD Mapping, Tuning, and Modeling

Mapping an application to CDs starts by building an abstract CD tree representation of the application and specifying the application-specific CD properties of each CD within the tree. CD properties are generally machine agnostic and include the length of the *compute* body, the volume of the preservation data, and the *recovery scope*. These properties can ideally be determined by a compiler or profiler from the CD-annotated source code but were extracted manually in this dissertation. After the initial mapping, the CD tree is still abstract and has not yet targeted a specific machine or a specific error model. The exact instantiation of the tree and the choice of appropriate preservation storage and preservation interval are determined by tuning the initial mapping. The tuning process takes into account the storage and bandwidth hierarchy, machine scale, and expected error model. Finding the optimal mapping of the application onto a machine is challenging because of the numerous optimization options offered by CDs and the difficulty of estimat-

ing expected performance. Fortunately, the concise abstractions of CDs are amenable to automatic tuning and optimization using the CD characteristics. Towards this end, we build an analytical model that takes a high-level description of an application structure and the underlying machine architecture to perform system efficiency estimation. The model enables quick and thorough exploration of the CD design space to determine trade-offs such as: (1) what level of storage should be used for each preservation; and (2) whether the CD hierarchy should be made deeper or more shallow to trade off localized recovery with preservation overheads (levels can be arbitrarily added or removed, so long as communication and synchronization semantics are preserved).

4.2.1 An Illustrative Example

Listing 4.1 shows an example mapping of the Sparse Matrix Vector (SpMV) multiplication kernel from the Parboil [79] benchmark suite. The API calls used in this example will be explained in detail in Section 4.3. The kernel has two levels of CD hierarchy; both levels have warp-scope recovery since there is no communication among warps. At the beginning of each CD, explicit preservation API calls are made to specify the input data and their storage location within the GPU memory hierarchy. Note that kernel input parameters are backed in special constant memory and the built-in thread indices such as *threadIdx.x* are stored in dedicated registers in NVIDIA GPUs, thus not requiring preservation.

Listing 4.1: Example CD mapping: SpMV

```
1  __global__ void spmv_jds_naive(float *dst_vector,
2      const float *d_data, const int *d_index,
3      const int *d_perm, const float *x_vec,
4      const int dim)
5  {
6    cd_begin(WARP);
7
8    // Kernel parameters are backed in constant memory and
9    // the built-in thread indices are stored in special
10   // registers thus not requiring preservation
11
12   int ix = blockIdx.x * blockDim.x + threadIdx.x;
13
14   if (ix < dim)
15   {
16     float sum = 0.0f;
17     int bound = sh_zcnt_int[ix / 32];
18
19     for(int k = 0; k < bound; k++)
20     {
21       cd_begin(WARP);
22       preserve_scalar_reg(k);
23       preserve_scalar_reg(sum);
24
25       int j = jds_ptr_int[k] + ix;
26       int in = d_index[j];
27
28       float d = d_data[j];
29       float t = x_vec[in];
30
31       sum += d * t;
32
33       cd_end(WARP);
34     }
35     dst_vector[d_perm[ix]] = sum;
36   }
37   cd_end(WARP);
38 }
```

In order to keep the overhead of preservation low, CDs should begin where the amount of live-in state is small. Unlike the previous approach by Menon et al. [57], which relies solely on compiler analyses in a best-effort manner, FGCDs provide an API-based programming construct to allow the programmer to tune the location and frequency of preservation using the analytical model based tuning framework discussed in Section 4.2.3. This flexibility makes it possible for FGCDs to adapt to a wide range of error rates without losing much efficiency. The API has several other advantages over the compile-time analysis in that; (1) identification of input array variables with potential clobber antidepences is easier and more exact (i.e. detecting WAR and RAW dependencies among memory addresses is difficult at compile-time, often resorting to conservative solutions [56].); and (2) high-level knowledge available at the source code level can be utilized to enable more efficient resilience mechanisms such as elimination of redundant preservation data among threads within the same CD.

Note that Listing 4.1 is an example mapping only and may not be optimal. Our analytical model based tuning framework is able to quickly evaluate efficiencies of vast variety of mapping options and find the optimal mapping. The main knobs in this tuning process are: (1) addition or removal of a CD level, (2) choice of the preservation storage, and (3) preservation interval for CDs within a loop (e.g. the inner CD in Listing 4.1 can be modified to span across multiple loop iterations trading off preservation overhead with recovery overhead.)

4.2.2 Model Assumptions

To simplify the analytical model and focus on the mapping and analysis of CDs, we make several assumptions. For the error model, we assume the presence of multiple independent fault/error processes that affect different aspects of the system. Each CD has an error rate associated with those errors it can locally recover (without escalation). We then associate these error processes with different levels of CD hierarchy. Finally, we assume that events within each error process are independently and identically distributed. The implication of these assumptions is that we can use a binomial model for CD failure and re-execution: the probability that a CD fails, p, is directly proportional to its run time and the sum of all error rates that it contains.

At the execution model level, we assume that the application forms a balanced tree hierarchy with no load imbalance. All errors associated with a CD are continuously detected within the body of the CD, and recoveries can be initiated immediately upon error detection. Preservation and restoration overheads are assumed to be symmetrical and are derived from the volume of preserved state and available storage bandwidths. We also assume that recovery does not overlap with normal execution (i.e. the extra time required for recovery is added to the time of a faulty CD). Recovery of sibling CDs, however, can proceed in an uncoordinated fashion in the absence of a synchronization or blocking communication.

4.2.3 Analytical Model

The analytical model estimates the execution efficiency (performance) of an application. Execution efficiency is defined as the percentage of runtime that the program is doing useful work (as opposed to additional work imposed by FGCDs such as preservation or recovery). The model accepts: (1) a description of the CD tree, which conveys crucial information such as the degree of parallelism, the locations of synchronization, the preservation overhead, and execution time of each CD, (2) a description of the target GPU architecture including bandwidth of each storage hierarchy and maximum throughput of each SIMD processor, and (3) an error model that determines the probability of failure in each CD.

The hierarchy allows us to look at only two levels (a parent and its children) of the tree at a time and recursively derive the overall system performance. Analysis starts with the lowest two levels, where children are leaves, and derives the impact of resilience on the parent. At that point, the parent execution properties are modified based on the preservation and recovery overheads associated with its children and the entire two levels are encapsulated. This process continues until the outermost, or the *root*, level is reached and the entire application properties are estimated. Due to this recursive process, we describe the model with respect to a parent with a set of child CDs. The development of the performance model takes the following three steps: (1) derive a model for a parent that has n identical children that all execute in parallel

with no sequential loops; (2) extend the model to include serial dependencies; and (3) allow sequential groups of CDs to be heterogeneous.

Parent with n identical parallel children: We start with a parent that has n identical parallel children. We first restrict ourselves to the case where the execution and recovery times for a particular child, T_c are uniform and do not account for execution variation. When a child fails, it is re-executed in full. During re-execution, the child may experience another error and may re-execute again. When n independent parallel children are grouped within the parent, the expected execution time of the parent is directly proportional to the expected maximum number of consecutive failures experienced by any one of the n parallel children.

Due to the model assumptions, the number of iterations of each CD follows a geometric random variable. While the statistics of geometric variables are well understood, we derive our model from first principles. Let $q[x, n]$ be the probability that all child CDs experience at most x consecutive failures. We then derive $d[x, n]$, the probability that the child with the most consecutive failures experiences exactly x failures and then succeeds. We derive $d[x, n]$ by subtracting the probability that fewer than x failures occur from the probability that at most x failures occur (thus leaving only the probability of exactly x failures). We use $d[x, n]$ to compute the expected run time of the parent. In all equations, we use p_c to represent the probability that a child fails; which we derive from the inherent error rate (p) associated with the child.

$$p_c = T_c p \tag{4.1}$$

$$q[0,n] = (1-p_c)^n \tag{4.2}$$

$$q[x,n] = \left(\sum_{i=0}^{x} \left(p_c^i \left(1-p_c\right)\right)\right)^n \tag{4.3}$$

$$d[0,n] = q[0,n] \tag{4.4}$$

$$d[x,n] = q[x,n] - q[x-1,n] \tag{4.5}$$

$$T_{parent}[x,n] = \sum_{i=0}^{\infty} (i+1)\, T_c d[i,n] \tag{4.6}$$

Serial dependencies between children: Next, we include the case where there are n parallel siblings, each responsible for executing m CDs sequentially. This can be thought of as having n warps running in parallel in a loop for m iterations. We follow the same derivation as above, but extend the definitions of the functions as follows. Let $q[x,m,n]$ be the probability that each of the siblings experiences at most x failures in the m serial children they contain. Similarly, $d[x,m,n]$ is the probability that the sibling with the most failures experiences exactly x failures before all of its m children succeed. This behavior is illustrated in Figure 4.3 and the model is shown below.

$$q[0,m,n] = (1-p_c)^{m \cdot n} \tag{4.7}$$

$$q[x,m,n] = \left(\sum_{i=0}^{x} \left(\binom{i+m-1}{i} p_c^i \left(1-p_c\right)^m\right)\right)^n \tag{4.8}$$

$$d[0,m,n] = q[0,m,n] \tag{4.9}$$

$$d[x,m,n] = q[x,m,n] - q[x-1,m,n] \tag{4.10}$$

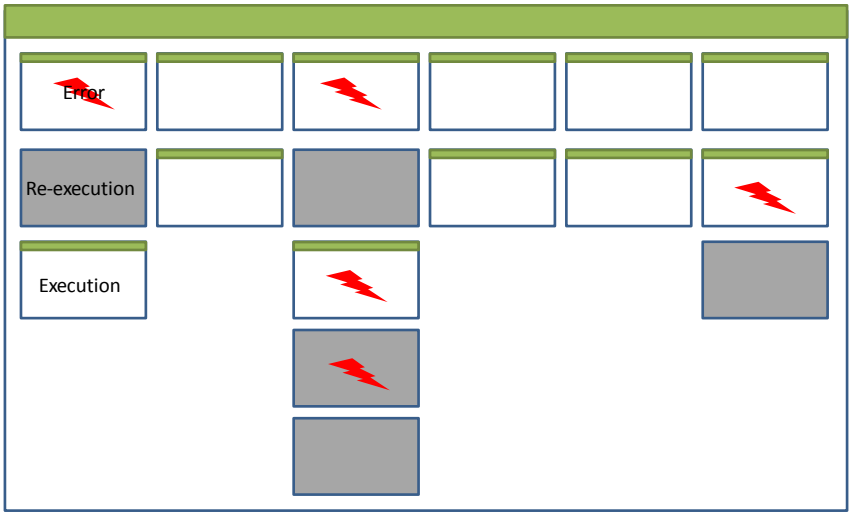


Figure 4.3: Example of 6 warps, each executing 2 sequential children. Re-execution of failed CDs can overlap if they have warp-scope recovery.

$$T_{parent}[x, m, n] = \sum_{i=0}^{\infty} (i + m)\, T_c d[i, m, n] \tag{4.11}$$

Heterogeneous CDs in serial: The model is extended to allow sequential CDs within a parent to differ. We rely on the assumption that all error processes are independent and derive the properties of an equivalent "average" child that can be directly substituted into the model for expected parent execution time. These properties are shown below assuming there are t different CDs, each with its own error model (these CDs execute sequentially, which allows us to generalize the error model easily). Note that because the re-execution time now depends on which CDs failed, we weight the average recovery time based on the execution time of each CD.

$$p_{c,avg} = \sum_{i=1}^{t} \frac{T_{c,i} p_i}{t} \tag{4.12}$$

$$T_{c,noerr} = \sum_{i=1}^{t} \frac{T_{c,i}}{t} \tag{4.13}$$

$$T_{c,rexec} = \frac{\sum_{i=1}^{t} T_{c,i}^2}{\sum_{i=1}^{t} T_{c,i}} \tag{4.14}$$

$$T_{parent}[x, m, n] = \sum_{i=0}^{\infty} \left(m T_{c,noerr} + i T_{c,rexec} \right) d[i, m, n] \tag{4.15}$$

4.3 FGCDs API

The FGCDs API provides an interface for programmers to specify CD properties and hierarchical tree structure of a GPU kernel and convey the information to the runtime and the hardware components of FGCDs. The API functions are implemented in NVIDIA CUDA C language. Number of custom defined NVIDIA PTX[3] instructions are introduced and used in the API function implementation as an inline assembly where interaction with the low level hardware is required. In this section, we discuss the specification and implementation of the FGCDs API functions.

4.3.1 CD Definition

As shown in Listing 4.1, a GPU kernel is mapped to a CD tree structure by defining the boundaries of CDs and assigning a recovery scope for each

[3]Parallel Thread Execution, an intermediate representation (IR) language for NVIDIA GPUs [80].

Listing 4.2: CD definition

```
1  typedef enum {
2    WARP,
3    THREAD_BLOCK,
4    GRID
5  } recovery_scope_t;
6
7  void cd_begin(recovery_scope_t rs)
8  {
9    // Note that the below inline assembly statement is
10   // written in pseudo-code style for readability
11   asm("cd.begin  rs, allocate_predreg;");
12 }
13
14 void cd_end(recovery_scope_t rs);
15 {
16   asm("cd.end;");
17 }
```

CD. CD boundaries are defined using a pair of functions *cd_begin* and *cd_end*. Each CD can be associated with only one scope of recovery, and this information is passed as an argument. Note that the argument has no effect for *cd_end* but is left for readability. *cd_begin* function also directs the compiler to allocate a predicate register for each of the warps in the CD. This predicate register is used to distinguish whether we are in normal execution mode or in error recovery mode to aid the preservation and restoration API functions, as described in Section 4.3.2.

These boundary functions are translated into a single PTX instruction as shown in Listing 4.2. The PTX instruction manipulates the hardware structure that keeps track of the *recovery program counter* (RPC) addresses

and the predicate register. The detailed operation of the instruction is given in Section 4.4 where we discuss the hardware implementation.

4.3.2 Preservation and Restoration

Semantically, as described in Section 4.1, preservation is performed at the *preserve* component and restoration is done at the *recover* component. In actual implementation, we define a single API function that performs preservation during normal execution and restoration during recovery. A predicate register is set by the hardware recovery mechanism, as explained in Section 4.4.1, to determine the correct execution path at runtime. For simplicity, the preservation and restoration API functions are designed to preserve/restore a single variable to/from a specific storage location. Thus, multiple calls are needed to preserve multiple variables to different locations. Below we describe the implementation of preservation and restoration API functions for scalar and array variables.

4.3.2.1 Scalar Variables

Scalar variables that need to be preserved for recovery can be automatically identified by the compiler through idempotence analysis. We design a compiler pass that runs on each CD to find all registers with potential *clobber antidependence* using the algorithm proposed by De Kruijf et al. [58]. This is not only convenient but also necessary because some registers with clobber

antidependence might not be exposed to the programmer at the source code level, and solely relying on API can lead to wrong results during re-execution.

While the identification of preserve variables is done by compiler analysis, the storage location is specified via an API call. FGCDs support two types of storage locations for preservation of scalar variables: register file and shared memory. The function prototypes are shown in Listing 4.3. Preservation to register file is performed by calling *preserve_scalar_reg*. This function hints the compiler to allocate a register and use a *mov* instruction to preserve the argument *var*. Using extra register space for preservation might have significant performance impact (1) if the application has high register pressure, or (2) if the the parallelism is limited by register usage. In such cases, the programmer can choose to preserve variables in the shared memory using *preserve_scalar_shmem*. This function will statically allocate shared memory space and use a *st.shared* instruction to preserve the argument *var*. Preservation to shared memory is especially useful when a variable has the same value for all threads within a warp because the preservation space can then be shared. A common example is a loop index variable that is not dependent on the thread ID. By setting the boolean argument *aggregate* to *true*, the programmer can hint the compiler to allocate one shared memory location per warp instead of per thread. Note that we do not define an API function for preserving a scalar variable to the local memory space. Our compiler pass is an IR (intermediate representation) level analysis and we assume the backend optimization will determine the best registers to spill to the local memory.

Listing 4.3: Preservation of a scalar variable

```
1 template<typename T>
2   void preserve_scalar_reg(T var);
3
4 template<typename T>
5   void preserve_scalar_shmem(T var, bool aggregate);
```

4.3.2.2 Array Variables

For array variables, separate functions are implemented for different memory spaces. We choose to only support preservation to the same memory space that the array variable belongs to because it is the most reasonable choice for all the benchmark programs we study. (e.g. Arrays in the local memory space are private to each thread, thus the aggregate size will be too large for the shared memory and storing them in the global memory has no benefit over storing them in the local memory because they are mapped to the same physical DRAM.) These functions take the address and size of the array as arguments as shown in Listing 4.4. Note that the current implementation only supports statically determined array sizes at kernel launch. This way, the storage for preservation can be statically allocated reducing the runtime overhead. This is not a significant restriction in practice because dynamic memory allocation on current GPUs still has high overhead and is not widely used in optimized kernels [81].

Listing 4.4: Preservation of an array variable

```
1 void preserve_array_global(void* var, size_t sz);
2 void preserve_array_shmem(void* var, size_t sz);
3 void preserve_array_local(void* var, size_t sz);
```

4.3.2.3 Recovery Behavior

The FGCDs API implements both preservation and restoration in one function. To determine the correct execution path at runtime, a predicate register is assigned to each CD and is set by the hardware recovery mechanism described in Section 4.4.1. Listing 4.5 is an example code snippet and the corresponding PTX instructions generated by our compiler pass. Upon entrance into the CD, a predicate register ($p200$ in this example) is assigned and reset by the *cd.begin* instruction. During normal execution, live-in variables i ($\$r4$) and *sum* ($\$r5$) are preserved and the program jumps to the *compute* body. When an error is detected, the predicate register $p200$ is set, and the preserved value is restored by jumping to the RESTORE label. Although the IR-level PTX code shown in Listing 4.5 uses branch instructions to manipulate the control flow, the backend compiler optimization can replace the branch instruction with predicated instructions if the number of instructions controlled by the branch condition is less than a certain threshold, which is either 4 or 7 instructions in current NVIDIA compiler [8].

A special case exists when an error is detected during preservation. If the preservation was incomplete, executing the restoration routine could lead to an undefined result. Thus, a special instruction *preserve.done* is inserted

Listing 4.5: Example PTX output

```
int sum = 0;
for (int i = 0; i < N; i++) {
  cd_begin(WARP);
  preserve_scalar_reg(i);
  preserve_scalar_reg(sum);
  ...
  sum += some_computation();
  ...
  cd_end(WARP);
}


// the above C code compiles to the below PTX

L1:
          cd.begin        WARP,  p200;
PRESERVE:
@p200     bra             RESTORE;
          mov.u32         $r201, $r4;
          mov.u32         $r202, $r5;
          preserve.done;
          bra             COMPUTE;
RESTORE:
          mov.u32         $r4, $r201;
          mov.u32         $r5, $r202;
COMPUTE:
...
          add.u32         $r5, $r5, $r7;
...
          cd.end;
@p0       bra             L1;
...
```

between preservation and restoration instructions to inform the hardware not to set the predicate on recovery so that the program performs preservation again.

4.4 Hardware Components of FGCDs

4.4.1 Hierarchical Recovery Support

As discuessed in Section 4.1, initiation of recovery is implemented in hardware. CD properties required for recovery are conveyed to the hardware by instructions *cd.begin* and *cd.end*. When the SIMD processor encounters these instructions, it updates the *recovery program counter* (RPC) stack, which is a per-warp stack structure maintained in hardware, with the information required for correct recovery. The *cd.begin* instruction pushes an entry on to the stack while the *cd.end* instruction pops the top entry. An RPC stack entry includes: (1) a 2-bit field to indicate the recovery scope of this CD, (2) a 32-bit field for the RPC, which is the program counter of the next instruction following the *cd.begin* instruction, (3) an 1-bit field which acts as the predicate register for this warp to distinguish whether we should perform preservation or restoration when encountered with a preservation API call as explained in Section 4.3, and (4) an 6-bit field which points to the current top of the SIMT stack to handle recovery of divergent warps. Figure 4.4 illustrates how the *cd.begin* instruction is executed in the processor pipeline. The stack push operation is performed at the DECODE stage by storing the recovery scope, the current SIMT stack top position, and the RPC address on to the top of the

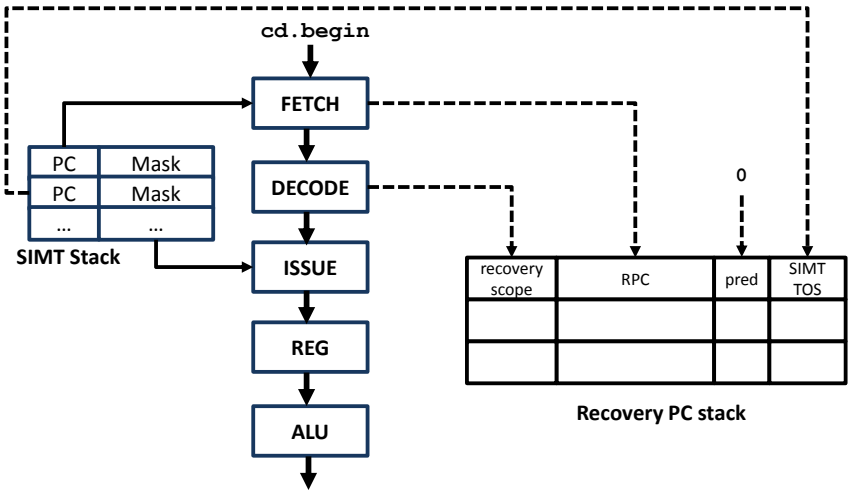


Figure 4.4: Execution of the *cd.begin* instruction.

RPC stack. The RPC address is obtained by reading the PC value of the next instruction from the FETCH stage. The predicate register is initially reset to '0' to indicate that we are in normal execution mode and thus should perform preservation instead of restoration.

The size of the RPC stack determines the maximum CD nesting degree that the programmer can specify, and three levels are enough for most applications. When an application exhibits a deeper degree of nesting, the programmer can find the best placement of CD boundaries by using our analytical model. The maximum number of warps that can reside in a SIMD processor is 48 in our evaluation, and thus the size of the recovery PC stack, assuming a 4-byte PC, a 2-bit recovery scope operand, and 1-bit predicate register, and a 6-bit SIMT stack pointer, is 738 bytes, which would amount to approximately 1.1% of the 64KB register file size. Another consideration is that operations

performed on this stack must be error-free. Since the circuit path is essentially a wire, we believe the stack operations should be very robust against timing errors, and extra protection can be implemented using *error correcting codes* (ECC).

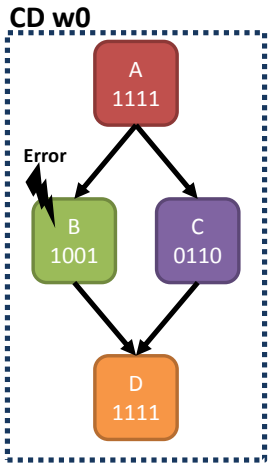
4.4.1.1 Initiation of recovery

When an error occurs, the error reporting architecture, as explained in Section 4.4.3, reports all potentially affected warps to the warp scheduler. The warp scheduler then reads the RPC stack of all affected warps and initiates an appropriate error recovery action depending on the recovery scope value. If the recovery scope of all erroneous warps is warp, error recovery is initiated by simply updating the PC of each warp with the RPC value and setting the predicate register value to '1' to jump to the restoration routine. Note that the predicate register is only set if the warp has already executed the *preserve.done* instruction. Otherwise, the predicate is left unset so that preservation routine is performed again on recovery. The recovery process needs to take extra steps if more than one erroneous warp has a recovery scope of thread block. First, the RPC stack top entries of all erroneous warps are checked to see if any of them belong to a thread-block-scope CD. If a warp belongs to a thread-block-scope CD, the RPC stacks of all other warps belonging to the same thread block are popped until they have the same top entry. Then each warp can roll back independently by reading its own RPC stack top entry. The hardware overhead should be small since the warp scheduler already has information

on which thread block each warp belongs to, and warps in the same thread block can be selected by a simple mask. An example of a thread-block-scope recovery is given in Section 4.4.1.2. Finally, for the recovery scope of grid, we assume a re-execution of the entire kernel and thus do not require modeling of the hardware.

4.4.1.2 Recovery of divergent warps

Threads within a warp can potentially diverge to different program paths if branch conditions do not match. NVIDIA GPUs handle this control divergence using a hardware stack, which we refer to as the SIMT stack, as explained in Section 2.2.1. When a divergent warp recovers from an error, the SIMT stack state must be restored to indicate the correct execution path with the correct active mask. Figure 4.5 shows an example of divergence within a warp scope CD. If a warp experiences an error while executing the divergent path B and rolls back to the convergent path A, the SIMT stack must discard any entries that have been pushed since the beginning of the CD and restore the PC of the re-executing path A. This can be easily implemented by recording the top of the SIMT stack (TOS) position into the RPC stack when executing the *cd.begin* instruction. Upon recovery, the SIMT stack is popped until the entry pointed to by the TOS field in the RPC stack becomes the top. Then, the PC of the current path is updated to the RPC value A, and the predicate register for this warp is set to '1' to jump to the restoration routine, as shown in Figure 4.5(b).



RPC stack

RS	RPC	P	TOS
W	A	0	0

SIMT stack

PC	Active mask	rePC
D	1111	-
C	0110	D
B	1001	D

(a) RPC stack and SIMT stack states before error. CD w0 has recovery scope (RS) of warp (W), RPC of A, predicate (P) register reset to '0', and the top of stack (TOS) pointing to the SIMT stack position '0' as shown by the arrow.

RPC stack

RS	RPC	P	TOS
W	A	1	0

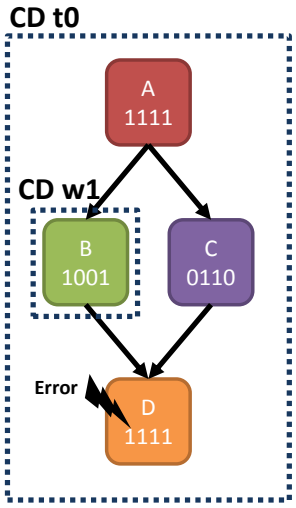
SIMT stack

PC	Active mask	rePC
A	1111	-

(b) When a warp experiences an error while executing basic block B, the SIMT stack is popped until position '0' becomes the top. The PC is updated to the RPC value A. The predicate is set to '1' to initiate restoration.

Figure 4.5: Recovery support for control divergence within a CD

The proposed mechanism for handling recovery of divergent warps not only supports divergence within a CD as shown in Figure 4.5, but also allows divergent threads to form a CD. For example, basic block B can form another level of CD as shown in Figure 4.6. In this example, the parent CD t0 has a recovery scope of thread block. Let's assume there are two warps (*warp1* and *warp2*) in CD t0, and an error occurs while *warp1* is executing basic block B, and *warp2* is executing basic block D. Since the top of the RPC stack for *warp2* has a recovery scope of thread block as shown in Figure 4.6(b), the RPC stack of *warp1* is first popped to have the same top position. Then, the SIMT stack of *warp1* is popped until the entry pointed to by the TOS field becomes the top. As a result, both the RPC stack and the SIMT stack of *warp1* and *warp2* have the same contents pointing to the recovery PC A.



RPC stack of warp1

RS	RPC	P	TOS
T	A	0	0
W	B	0	2

SIMT stack of warp1

PC	Active mask	rePC
D	1111	-
C	0110	D
B	1001	D

RPC stack of warp1

RS	RPC	P	TOS
T	A	1	0

SIMT stack of warp1

PC	Active mask	rePC
A	1111	-

(a) RPC stack and SIMT stack of *warp1* before (left) and after (right) error. CD t0 has recovery scope (RS) of thread block (T) and CD w1 has recovery scope of warp (W). *Warp1* is assumed to be executing basic block B when error occurs.

RPC stack of warp2

RS	RPC	P	TOS
T	A	0	0

SIMT stack of warp2

PC	Active mask	rePC
D	1111	-

RPC stack of warp2

RS	RPC	P	TOS
T	A	1	0

SIMT stack of warp2

PC	Active mask	rePC
A	1111	-

(b) RPC stack and SIMT stack of *warp2* before (left) and after (right) error. *Warp2* is assumed to be executing basic block D when error occurs.

Figure 4.6: Recovery support for diverged threads forming a CD

Note that divergence is not supported across CDs. i.e. Active threads at CD entry must be the same as active threads at CD exit.

4.4.2 Error Containment Mechanisms

Due to their tunable nature, FGCDs do not need to be tied to a specific error detection mechanism and can incorporate any type of error detectors with arbitrary latency as long as errors are contained within the CD that experiences them. To guarantee error containment while supporting arbitrary

error detection latency, communications across containment domains must be either (1) delayed or (2) buffered until verified to be free of errors by the error detector, or such communications must not propagate beyond a certain subtree within the CD hierarchy so that (3) escalation can be performed in case an error escapes from the current CD.

In general, two types of instructions can lead to violations of the error containment semantics: store instructions and CD boundary instructions. Here, store instructions include any instructions that can have side effects on memory: e.g. atomic operations. Store instructions can either write erroneous data to a shared location or write to an erroneous address; e.g., an address another CD reads or writes, or an address storing preserved state. Note that we break CD boundaries when correctly committeed stores may change the result of the program if re-executed, and thus only have to guaranteee correctness of stores before committing them. On the other hand, the CD boundary instruction *cd.end* can discard the context of a CD that has not yet been verified, effectively losing the ability to recover that CD. In this section, we discuss three mechanisms that can be used to guarantee error containment and explain how they apply to these two types of instructions.

4.4.2.1 Instruction Stall

The most straightforward way of providing error containment guarantees is to stall any instruction that can potentially communicate with other CDs until it is verified by the error detector. This mechanism can be applied

to both store and CD boundary instructions by adding per-warp counters that are set to the value of the error detection latency when a store or a *cd.end* instruction reaches its commit stage in the pipeline. The counters are then decremented each cycle until they reach zero. While a warp's counter has a non-zero value, the warp scheduler does not issue a new instruction from that warp to prevent propagation of unverified data. While being simple, the instruction stall mechanism adds extra latency to all communicating instructions and it can negatively affect performance. However, as will be discussed in Section 6.2.2, a few cycles or even a few tens of cycles of extra latency can be effectively hidden by the massive multithreading of GPUs.

4.4.2.2 Store Buffer

Depending on the kernel characteristics, stalling all store instructions can sometimes lead to a large performance penalty. For such cases, an extra store buffer structure can be used to let warps continue making progress without stalling. We assume a store buffer similar to the one used by Feng et al. [56] and Menon et al. [57], to delay committing store instructions until both the address and the data of the store instruction are verified to be correct. An important design decision is choosing the right size of the store buffer. The store buffer must be large enough to provide performance benefits over stalling, but small enough to keep the area and power overheads small. Our baseline SIMD processor architecture described in Table 5.1 has two warp schedulers, but only one scheduler can issue a store instruction on any given

cycle. With a 32-bit address space, a single store instruction pushes 256 bytes of data into the store buffer; i.e., 32 threads saving 4 bytes of data and 4 bytes of address each. Given that the L1 cache size is 16KB and that the maximum reasonable size of the store buffer should be at least an order of magnitude smaller than that, we choose 2KB in our evaluation, which translates to an 8-entry store buffer.

4.4.2.3 CD Escalation

Both delaying (instruction stall) and buffering (store buffer) try to contain potentially erroneous data within the current CD so that in case of an error, the failing CD can recover locally without coordinating with other CDs. As a result, we end up paying constant performance and power overheads even when not experiencing any errors. Instead of over-designing for the worst case, we can optimize for the common case of error-free execution by taking advantage of the escalation capability discussed in Section 4.1. With escalation, instructions are speculatively committed assuming that they will be error free, and in case an error escapes the current CD, recovery is escalated to the parent CD. Since we need to guarantee that the preserved state of the parent CD is not corrupted by the error, store instructions, which can potentially write to arbitrary addresses, still need to be stalled or buffered while CD boundary instructions can be speculatively committed. To support escalation of CDs that have crossed the CD boundary, each warp is assigned with a hardware counter which is set to the value of the error detection latency when the warp

executes the *cd.end* instruction. Similarly to the case of instruction stall, these counters are decremented each cycle, and when an error is detected, all warps with a non-zero counter value pop the top entry of the RPC Stack discussed in Section 4.4.1 to escalate the recovery to the parent CD.

4.4.3 Error Reporting Architecture

Modern commodity processors from Intel (e.g., Pentium 4, Xeon, Itanium), AMD (e.g., Opteron), and IBM (e.g., Power-7) incorporate a *machine check architecture* (MCA) [82, 83, 84, 85] to contain and report errors that are detected by hardware mechanisms. Errors are reported through dedicated machine-check registers that hold logged information about errors on different parts of the system, or through data poisoning which taggs corrupted data with poison bits. Notification of errors is generally done with machine-check exceptions that target the operating system in current systems.

GPUs are different from traditional CPUs in that; (1) they have a massive number of concurrent threads that can be affected by an error, and (2) they do not have general exception support. These GPU-specific aspects necessitate a different error reporting architecture. First, the massively threaded nature of GPUs means that multiple warps could have been scheduled on the pipeline while the pipeline was in an unstable state. In order to identify the warps that are affected by an error, the error reporting architecture keeps a list of warps that were active for the last n cycles, where n is the latency of the error detector. Note that the warp scheduling policy has an impact on

the number of affected warps. Recent proposals [86] on GPU warp scheduling suggest that variants of *greedy-than-oldest* (GTO) scheduling policy performs well across different applications, and such a policy can help reduce the number of affected warps by favoring instructions from a single warp until it stalls. Second, due to the lack of general exception support, the error reporting is done by notifying the warp scheduler of those potentially affected warps. The warp scheduler then initiates error recovery by forcing control to jump to the registered RPC as explained in Section 4.4.1.

In this dissertation, we assume that error detection is done at the granularity of a SIMD processor, and thus incorporate the proposed error reporting architecture for each SIMD processor.

Chapter 5

Methodology

In this chapter, we discuss the methodologies we use to evaluate the applicability of fine-grained containment domains to timing speculation on GPUs. We first explain the error and fault model that we assume in our evaluation. This model gives the relationship between the operating voltage and the error rate which is crucial in determining the efficiency of timing speculation. The simulation infrastructure and the properties of the simulated GPU system are described next, followed by a description of the GPGPU workloads that are studied in the evaluation. Finally, we explain how the analytical model is used to optimize the CD mappings and validate the model against simulation results.

5.1 Simulation Model

The microarchitectural components of FGCDs are modeled using GPGPU-Sim [87], which is a detailed cycle-level performance simulator for a general purpose GPU architecture. GPGPU-Sim can simulate either the PTX instruction set, which is an intermediate language used by NVIDIA to target different hardware generations, or the SASS (Shader ASSembly) instruction set, which

Table 5.1: Simulator configuration.

Number of SIMD processors	15
SIMD processor clock frequency	700MHz
Number of threads / SIMD processor	1536
Number of warps / SIMD processor	48
Number of thread blocks / SIMD processor	8
Warp size	32
SIMD pipeline width	32
Number of warp schedulers / SIMD processor	2
Warp scheduling policy	Greedy-then-oldest [86]
Registers / SIMD processor	32768
Shared memory size / SIMD processor	48KB
L1 cache (size/associativity/block size)	16KB/4-way/128B
L2 cache (size/associativity/block size)	768KB/16-way/128B
Number of memory channels	6
Memory bandwidth	177.6 GB/s
Memory controller scheduling policy	Out-of-order (FR-FCFS)

is the native instruction set of NVIDIA GPUs. The PTX representation does not incorporate important compiler optimizations such as instruction scheduling and register allocation, and thus cannot accurately model the preservation overhead imposed by FGCDs. Therefore, we choose to simulate the SASS instruction set in our evaluation.

We configure GPGPU-Sim to closely match NVIDIA's GTX480 [4] using the configuration file provided with the simulator [88]. Table 5.1 lists the key microarchitectural parameters of the simulated GPU system.

5.2 Benchmarks

We study the applicability of FGCDs to timing speculation using select applications from Parboil [79], LonestarGPU [89], and the benchmarks

provided with GPGPU-Sim [87]. These benchmark suites represent commonly used GPGPU workloads with different application characteristics and are widely adopted in the academia for GPGPU research. Table 5.2 shows the list of benchmark programs that are selected for evaluation. The table lists some of the key application characteristics that affect the efficiency of FGCDs. The total number of threads and the instruction count show the amount of parallelism as well as the average amount of work done by a single thread. The presence of inter-warp communication means that a thread-block-scope CD might be required, and the presence of global communication (inter-thread-block communication) indicates that a grid-scope CD might be needed. For the purpose of timing speculation, applications that have many threads with large amount of work and no communication generally benefit the most from FGCDs. This is due to the fact that (1) lack of communication allows localized recovery, and thus performance efficiency can remain high even at high error rates (or lower supply voltage), and (2) having a large amount of work per thread usually lead to more optimization options when trading off preservation overhead with recovery overhead.

In addition to the application characteristics shown in Table 5.2, other aspects of applications such as register pressure, memory bandwidth requirement, and regularity of control also have significant impact on how well the application performs with FGCDs. Below we describe each benchmark in more detail with these characteristics in mind, and give the reasoning behind the inclusion of the benchmark in our evaluation.

Table 5.2: Benchmarks studied for evaluation.

Name	Description	Total threads	Inst. count	Warp comm.	Global comm.
AES [87]	Advanced Encryption Standard	65792	28M	Yes	No
LIB [87]	LIBOR Monte Carlo	4096	907M	No	No
LPS [87]	3D Laplace solver	12800	82M	Yes	No
CP [79]	Coulombic Potential	32768	126M	No	No
SPMV [79]	Sparse matrix vector product	146880	78M	No	No
BFS [89]	Breadth first search	1070592	195M	No	Yes
SSSP [89]	Shortest path	1070592	265M	No	Yes
SP [89]	Survey propagation	20480	8M	No	No

AES AES is an implementation of the Advanced Encryption Standard by Manavski [90], and we encrypt a 256KB picture using 128-bit encryption. The algorithm is clearly divided into eleven stages of computation where each stage reads from the shared memory, computes on the data, and writes the results back to the shared memory for the next stage. Explicit barriers are used at the end of each stage to synchronize accesses to the shared data among threads within a thread block. This benchmark is interesting because we could either form a warp-scope CD for each stage for localized recovery or form a thread-block-scope CD for all eleven stages to trade-off recovery overhead with preservation overhead.

LIB LIB performs Monte Carlo simulations on the London Interbank Offered Rate Market Model [91]. The kernel has three levels of nested loops where the innermost loop can be mapped to a warp-scope CD. The amount of work done by a thread is also large as compared to other GPU kernels. This allows us to tune the preservation interval using our analytical model based

performance estimation. The innermost CD calls both *preserve_scalr_reg* and *preserve_array_local*.

LPS LPS implements the 3D Laplace solver, and we run one iteration on a 100x100x100 grid. The main loop in this kernel iterates for 100 times (the Z-dimension size of the input grid) and maps to a thread-block-scope CD since threads communicate via shared memory. The amount of shared memory usage is relatively small (1,944 bytes) and can be preserved by calling *preserve_array_shared*.

CP CP calculates the electrostatic potential field produced by charged atoms distributed throughout a volume [79]. The amount of work per thread depends on the number of atoms simulated, and we simulate 4000 atoms on a grid size of 256x256. This application is highly regular with low bandwidth requirement.

SPMV This kernel implements the Sparse Matrix Vector product with clearly isolated input and output arrays as shown in Listing 4.1, and the absence of read-write arrays makes the entire kernel idempotent. SPMV has high bandwidth demand and there is a load imbalance between threads potentially causing control divergence.

BFS BFS is a classic graph traversal algorithm which traverses the input graph and labels each node with the distance from a designated source node assuming unit weight edges. The algorithm assigns each node in the input

graph to a single thread, and each thread visits all its neighbors to update the distance. Since the number of neighbors vary among nodes, the control flow is irregular. Updating of the distance is done using atomic operations, and thus can be seen as a form of global communication. The input graph we use represents the USA road network with 1M nodes and 2.7M edges.

SSSP SSSP calculates the shortest path of each node from a designated source node in a directed graph with weighted edges. SSSP is very similar to BFS but exhibits extra irregularity for the same input graph due to the weighted edges.

SP SP is a heuristic SAT-solver based on Bayesian inference [92]. This application has relatively high bandwidth requirement while having multiple global arrays that needs to be preserved. Since each thread only performs a few hundreds of instructions, the overhead of preservation can limit the efficiency of FGCDs.

5.3 Model-Based Optimization Methodology

This section describes how the analytical model is used to derive the optimal CD mapping for a given application. The optimization objective we use throughout this dissertation is energy consumption. However, the methodology is not limited to energy optimization and can be applied to other metrics such as the energy-delay-squared product (ED^2P) [93].

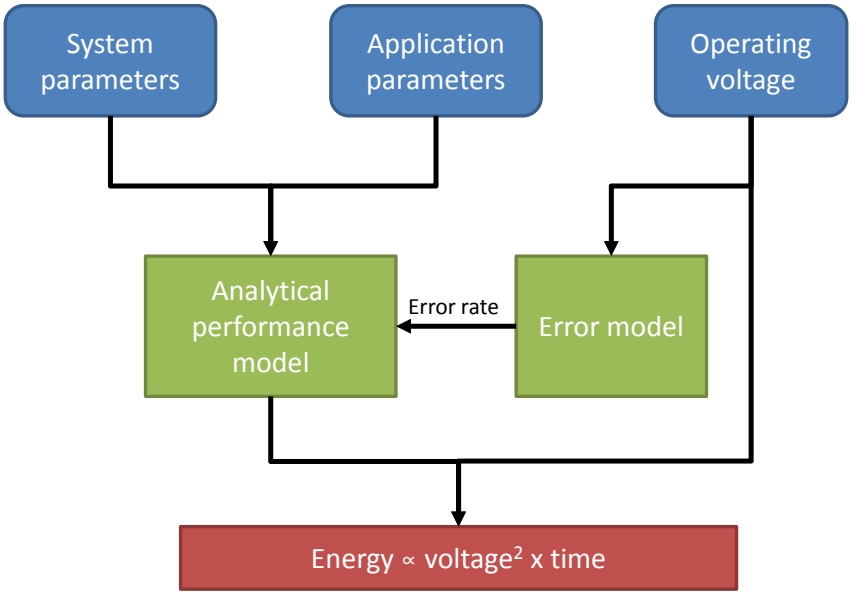


Figure 5.1: Model-based optimization methodology

5.3.1 Overall Methodology

Figure 5.1 shows the overall methodology of obtaining the energy efficiency of an application for a given set of parameters at a given operating voltage. For each operating voltage point, the corresponding error rate is given by the error model. Then, for each error rate value, the CD properties are tuned to find the parameters that lead to the highest performance efficiency. The tuning process is an exhaustive search where all possible combinations of preservation frequencies at each level of CD hierarchy is tested. For example, a CD can be tuned to span multiple loop iterations to reduce the overhead of preservation at the cost of potentially longer recovery.

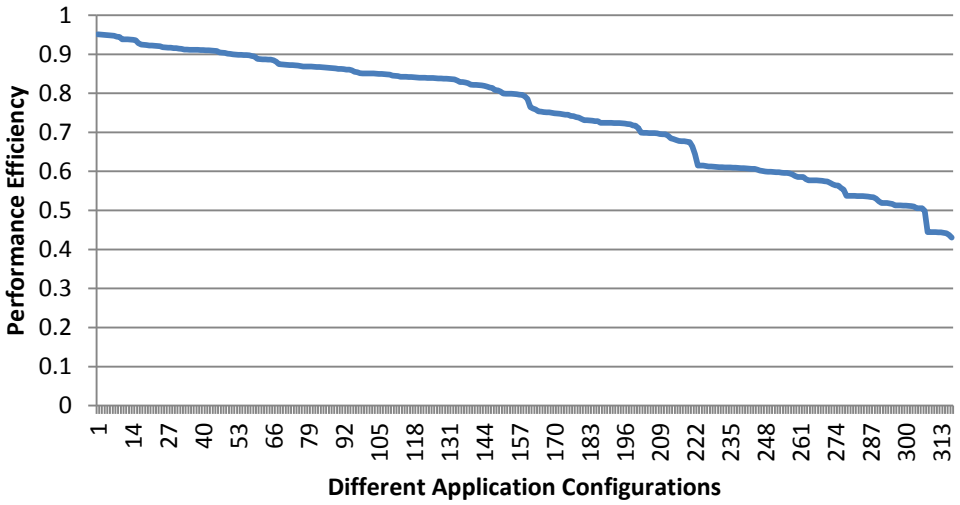


Figure 5.2: Tuning the Matrix Multiplication kernel running at Vdd=0.93V

Figure 5.2 shows an example optimization space exploration at a fixed voltage ($0.93V$) for the Matrix Multiplication kernel shown in Figure 4.2. More than 300 different CD mappings are evaluated and depending on the mapping, the performance efficiency can vary from 40% to 95%. The same process is repeated for a range of operating voltages to find the best voltage and CD mapping pair.

Modeling of GPU System Parameters: According to Table 5.1, the simulated GPU system has 15 SIMD processors with a 32-wide pipeline that can issue two instructions per cycle, leading to a maximum IPC of 960. Since a SIMD processor can execute 64 thread instructions per cycle, we assume that a thread block mapped to the SIMD processor can run 64 threads in parallel and execution of more threads will be serialized. Similarly, at most 15 thread

blocks can run in parallel and execution of more thread blocks is assumed to be serialized.

For the preservation bandwidth, we use the maximum theoretical bandwidth of each storage location as follows: the register file provides 4 bytes/cycle to each thread, the shared memory provides 128 bytes/cycle to a SIMD processor, and the DRAM bandwidth of 177 GB/sec is assumed to be shared by 960 threads running at 700 MHz.

Extracting Application Parameters: The structure of the CD hierarchy, the preservation volume for each CD, and the expected length (in number of cycles) of each CD need to be fed into the analytical model to estimate the efficiency of the application for a given error rate. Mapping the application to FGCDs using the proposed API implicitly gives the CD hierarchy and the preservation volume for each CD level. The expected length of a CD, on the other hand, is derived through profiling using GPGPU-Sim. We derive the expected number of cycles by multiplying the dynamic instruction count by the average *instructions per cycle* (IPC).

Adjusting the error rate: Due to the massively threaded nature of GPUs, multiple warps and thread-blocks are simultaneously resident on a SIMD processor, time-sharing the pipeline. Since a voltage droop event can corrupt any computation that is happening in the pipeline, a single error can potentially affect multiple independent CDs, effectively increasing the error rate that each

CD experiences. To account for this effect, we profile the average number of cycles a CD has at least one active instruction in the pipeline (and thus is vulnerable to errors), and adjust the error rate using equation 5.1.

$$\text{CD error rate} = (\text{SIMD processor error rate}) \times \frac{\text{Vulenrable cycles}}{\text{CD execution time in cycles}} \tag{5.1}$$

5.3.2 Analytical Model Validation

In this section, we validate the analytical model by comparing the model results with the simulation results across a range of different CD mappings and operating voltages. Since the analytical model is developed to aid the optimization process, the most important quality of the model is how well it follows the simulation trend in relative terms rather than how close the absolute numbers are. To show the validity of the analytical model for optimization purposes, we present three different kernels having different characteristics. First, we present the simple case of adding or removing a CD level in the AES kernel. Next, effect of varying preservation intervals is studied for the LPS kernel. Lastly, the BFS kernel is selected to show the model accuracy for kernels with irregular control and data access.

5.3.2.1 Model accuracy across different CD mappings

We first take two different CD mappings of the AES kernel and estimate the relative performance efficiencies at varying operating voltages using both

the analytical model and the simulator and show the results in Figure 5.3. The simulation results represent average of ten random error injection experiments with a 95% confidence interval shown with error bars. Each thread in the AES kernel is relatively short-lived executing about 300 instructions. We first map the kernel to a 2-level CD hierarchy where the parent CD is a thread-block scope CD containing all 300 instructions, and the child CDs are warp-scope CDs containing a few tens of instructions each, with a thread-block-wide barrier instruction at the end. There are eleven child CDs that are executed sequentially, each preserving several registers. Since these child CDs incur extra preservation overhead, we also try a 1-level CD mapping by removing the child CDs. These two CD mappings are chosen to demonstrate the trade-off between the preservation and the recovery overhead at different operating voltages and show how well the analytical model tracks the simulation results in such an optimization space.

Figure 5.3 shows that overall the model trend closely follows the simulation results across a wide range of operating voltages. The model does tend to overestimate the efficiency, especially at lower voltage (higher error rate). This is due to the fact that frequent recovery causes excessive constant memory bank conflicts, negatively affecting the SIMD processor throughput during simulation while the model assumes a constant IPC for all operating voltages. The discrepancy only becomes prominent when the error rate is very high and such extreme cases are far from the optimal operating voltage in all our experiments.

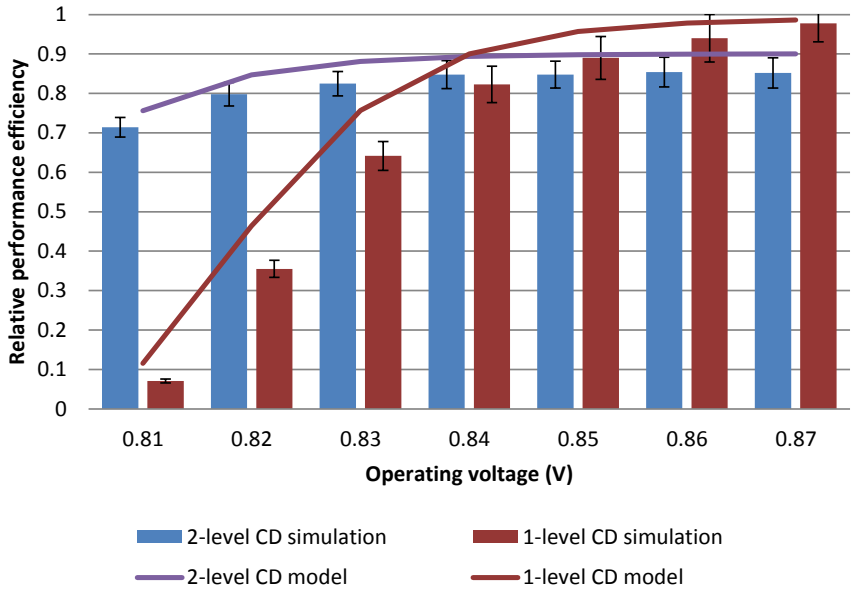
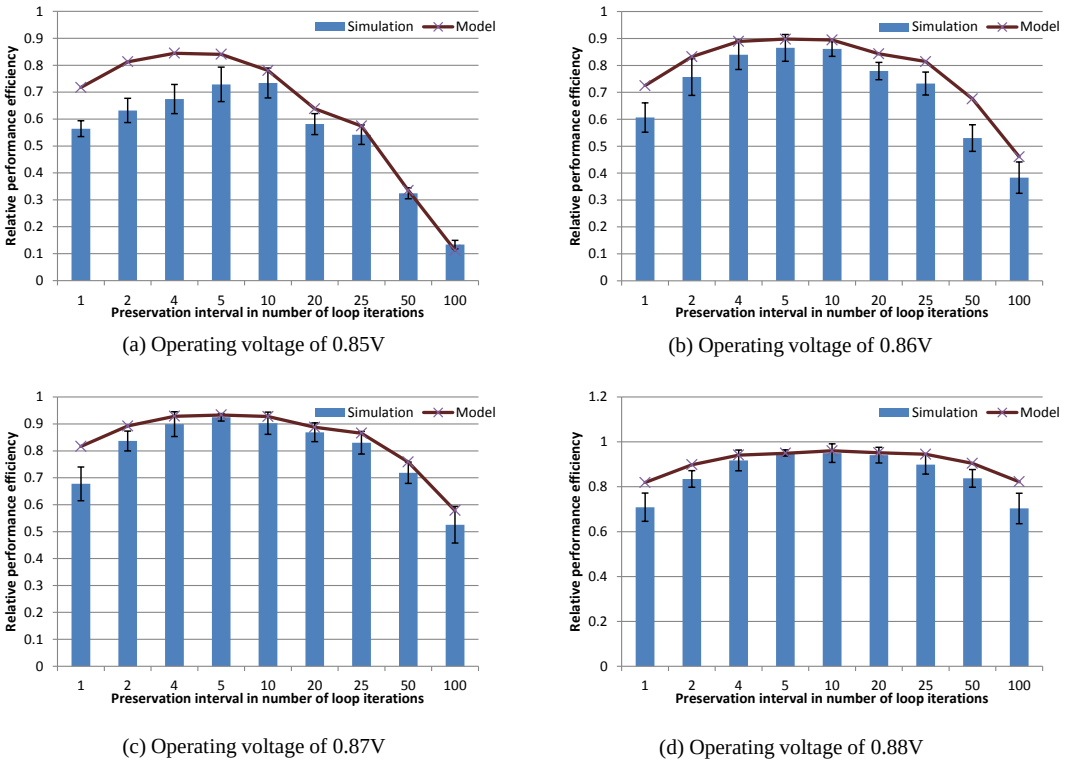


Figure 5.3: AES kernel efficiency estimation using both the analytical model and the simulator. Error bars represent 95% confidence interval.

5.3.2.2 Model accuracy across varying preservation intervals

For many applications, the main optimization knob for FGCDs is the preservation interval. We take the LPS kernel as an example and show that our analytical model accurately estimates the performance efficiency of an application with varying preservation intervals. The kernel loops over the z-direction of a 3D space and performs Jacobi iterations on each of the x-y planes. Each loop iteration updates the plane data array stored in shared memory, requiring preservation. The size of the input grid we use is 100x100x100, and thus 100 iterations are run. Preserving the input data array for every single iteration would incur high preservation overhead but would make the kernel more tolerant to high error rates. Conversely, preserving only once for all 100



(a) Operating voltage of 0.85V

(b) Operating voltage of 0.86V

(c) Operating voltage of 0.87V

(d) Operating voltage of 0.88V

Figure 5.4: LPS kernel efficiency estimation using both the analytical model and the simulator. Error bars represent 95% confidence interval.

iterations would incur negligible preservation overhead but would suffer from excessive recovery overhead if the error rate is high.

Figure 5.4 shows the estimated performance efficiency for a range of preservation intervals. The line graphs represent the model results and the bar graphs represent average values of ten random error injection experiments. The error bars are drawn to show a 95% confidence interval for the simulation results. The comparison is performed at four different operating voltages to show the sensitivity of the model to the error rate. Overall, the model trend

closely follows the simulation results as shown in Figure 5.4. Slight overestimation observed across all comparison points is the result of extra instructions introduced in preservation API functions and loop index calculations which the model does not take into account and is small enough to be safely ignored for the purpose of optimization. On the other hand, noticeable discrepancies can be seen when the performance efficiency is low. Similarly to the case of AES, excessive recovery (which can be inferred from the low performance efficiency) causes lower IPC during simulation while the model assumes a constant IPC. For LPS, this is mainly due to increased memory traffic leading to more memory latency related stalls in the pipeline. The LPS kernel shows a larger discrepancy than the AES kernel because it has a larger preservation volume which consumes the load/store bandwidth of the SIMD processor.

5.3.2.3 Model accuracy for irregular applications

Certain GPU applications experience a large degree of control or memory divergence, making it difficult to predict the performance with a model based approach. Our analytical model relies on profiling to extract key application characteristics such as the average IPC and the average number of cycles a warp or a thread block is vulnerable to errors. As discussed in Section 5.3.1, these numbers are used to derive the average runtime and error probability of a containment domain which are then fed into the model.

Figure 5.5 demonstrates how well our approach works for an irregular kernel. Each thread in the BFS kernel is assigned a node in the input graph

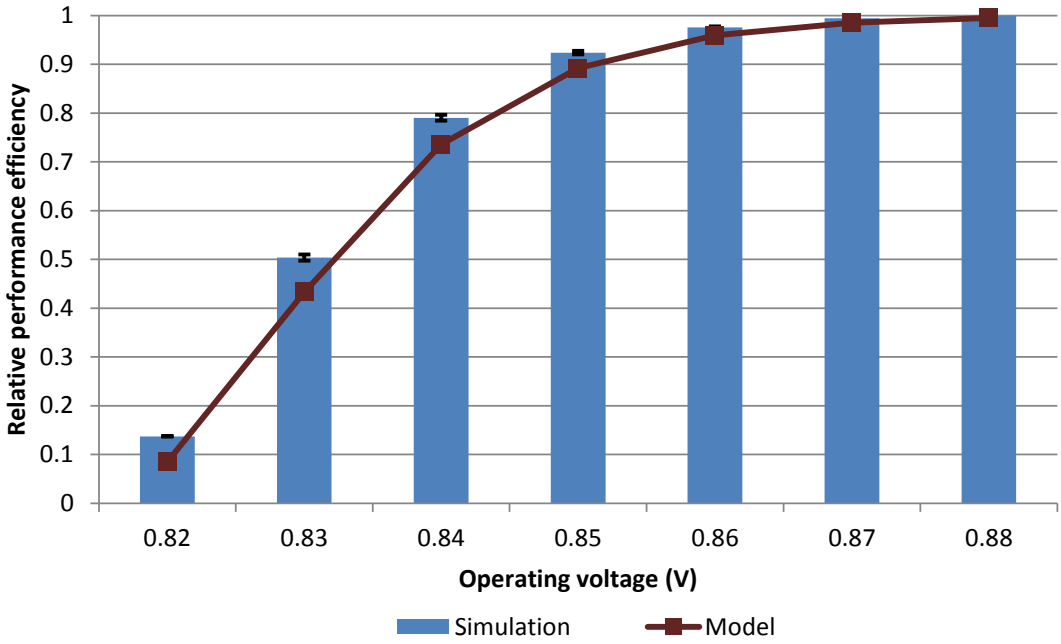


Figure 5.5: BFS kernel efficiency estimation using both the analytical model and the simulator. Error bars represent 95% confidence interval.

and loops over all neighbors of the node to find the shortest distance from a given root node. Nodes in the input graph we use have relatively low degree (only three neighbors on average), and each thread executes about 100 instructions on average. This leads to a simple 1-level CD mapping where the entire kernel is mapped to a warp scope CD. Again, the model closely follows the simulation result. One notable aspect of the BFS kernel is that diverged threads are always converged back on re-execution, increasing the effective IPC of the SIMD processor. Since the model uses a fixed IPC that is profiled without injecting any errors, this leads to underestimation of performance at low voltage as shown in Figure 5.5.

5.3.2.4 Model validation summary

The above examples show that our analytical model estimates the performance efficiency of an FGCD-enabled application with high accuracy and thus is a good tool for quickly exploring the diverse CD mapping options. Due to the simplicity of the model, discrepancies do exist and the main source of them is the IPC variation experienced during re-execution. Fortunately in our experience, large deviations only occur for CD mappings that are far from the optimum. To further ensure that the inaccuracies in the model do not lead to a suboptimal solution, we choose multiple CD mappings surrounding the optimal mapping and compare their simulation results.

Chapter 6

Experiments

This chapter provides a detailed evaluation of the fine-grained containment domains framework to show its applicability towards timing speculation. We first explore diverse CD mapping options for each workload using the analytical model based optimization methodology as discussed in Section 5.3. Based on the optimization results, potential optimal CD mappings and their corresponding operating voltages are identified. Detailed analyses are performed for these candidates with cycle-based simulation. Specifically, we present the energy savings, performance overheads, and sensitivity to error detection latency. In all graphs we show, energy consumption and performance overhead are relative to the error-free execution on the baseline architecture without FGCDs. As discussed in Section 3.1.1, the baseline architecture is assumed to run at 1.0V without generating any errors.

6.1 Model-Based Optimization

Each of the benchmarks described in Section 5.2 is mapped to FGCDs by examining the source code, annotating the source code with appropriate FGCDs API calls, and profiling key CD parameters using the GPGPU-Sim

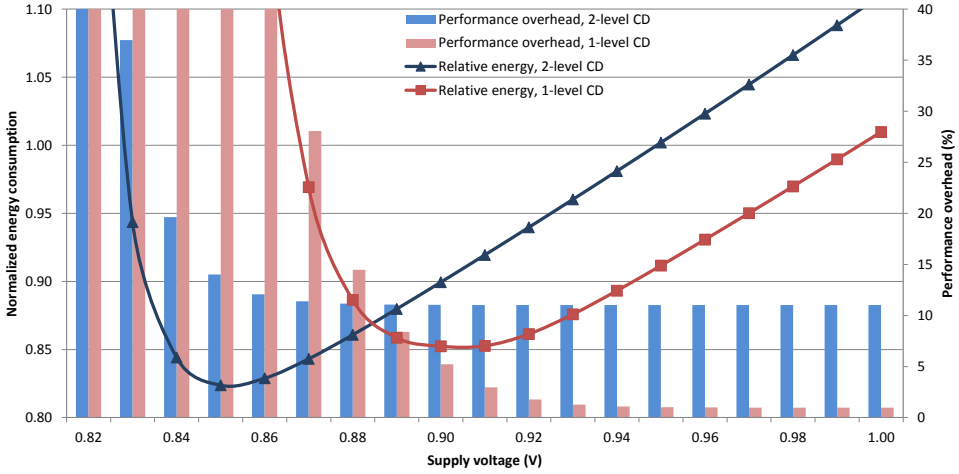


Figure 6.1: Optimal CD mapping and supply voltage for AES

simulator. The analytical model then estimates the relative performance overhead and energy efficiency of a vast number of possible CD mapping options across a range of operating voltages. Since our goal is to minimize the energy consumption, we find the CD mapping and the operating voltage pair that yields the lowest energy consumption.

6.1.1 AES

The AES kernel is a good example to show the simplest form of optimization: adding/removing a CD level to/from the CD hierarchy. As explained in Section 5.2, the algorithm goes through eleven stages of computation, and communication only occurs between stages via shared memory. Thus it's natural to map the entire kernel to a thread-block-scope CD and each stage to a warp-scope CD. Each warp-scope CD contains 9-43 instructions and preserves 0-4 registers. Since the overhead of preservation and extra CD instructions

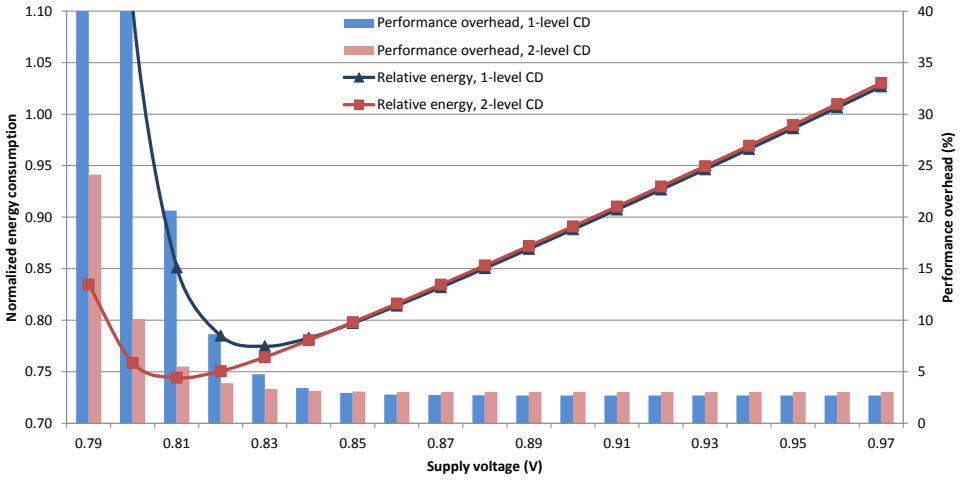


Figure 6.2: Optimal CD mapping and supply voltage for SP. The parent CD has four global variables which are preserved in the register file.

is relatively large for such short CDs, it is possible that removal of the warp CD level might result in better energy efficiency. Thus we estimate the normalized energy consumption of both mappings across a range of operating voltages (Figure 6.1). As expected, the two-level CD mapping results in lower energy efficiency when the error rate is low (supply voltage of 0.88V or higher). However, due to being able to recover locally within a stage, it outperforms the one-level CD mapping at higher error rates, giving the optimal energy point at 0.85V. Note that the bar graphs show the performance overhead of each mapping, and that the energy optimal point at 0.85V actually has about 10% performance penalty. A different metric of interest such as energy-delay product could lead to a different optimal point with a different CD mapping.

6.1.2 SP

SP is another example where the trade-off between preservation and recovery can be played by removing a CD level. Unlike AES where only the child CDs have large preservation overhead, the main computation kernel for SP requires both the parent CD and the child CD to preserve some data. Each thread in the parent CD preserves four global array elements, and each thread in the child CD preserves two register variables. The global array variables in the parent CD can be preserved to global memory; however, due to the relatively small number of threads the kernel has (20,480 as shown in Table 5.2), they can also be preserved to either the shared memory or the register file without hurting parallelism much. Figure 6.2 shows the best performing option where each thread preserves the global variables to the register file. As illustrated, two CD mappings behave similarly when the error rate is low. This is due to the fact that the preservation overhead is dominated by the parent CD and thus the two-level CD mapping does not suffer much from having extra preservation in the child CD. It is also interesting to note that the parent CD in SP is a warp-scope CD and thus can survive a higher error rate than AES with localized recovery when the one-level CD mapping is used.

6.1.3 CP

Adding or removing a CD level is a simple but restricted way of trading off between preservation overhead and recovery cost. In this section, we

Listing 6.1: Tuning preservation interval: CP

```
1
2 for (int i=0; i<N_ATOMS; i++) {
3   if (!(i%2)) cd_begin(WARP);
4
5   float dx = coorx - atom[i].x;
6   float dy = coory - atom[i].y;
7   float r = 1.0f / sqrt(dx*dx + dy*dy + atom[i].z);
8   energy += atom[i].w * r;
9
10  if (i%2) cd_end(WARP);
11 }
```

present a more generalized knob for the same trade-off, namely the preservation interval. A CD mapping with a long preservation interval leads to lower preservation overhead during error-free execution, but incurs higher recovery overhead when errors occur. Conversely, a CD mapping with a short preservation interval has extra preservation overhead to reduce the recovery cost. This trade-off becomes more apparent for CDs within a loop body. Listing 6.1 is a code snippet from CP's main iteration loop where each thread iterates through all atoms in the input grid to calculate the energy potential at each grid point. The loop body is a warp-scope CD that preserves two register variables i and *energy* which are identified using the idempotence analysis described in Section 2.5.2.2. The preservation interval (in number of loop iterations) for this warp-scope CD can be anywhere between one – meaning that we are preserving every loop iteration, and N_ATOMS – meaning that we are preserving only once during the first iteration of the loop. Two *if* statements

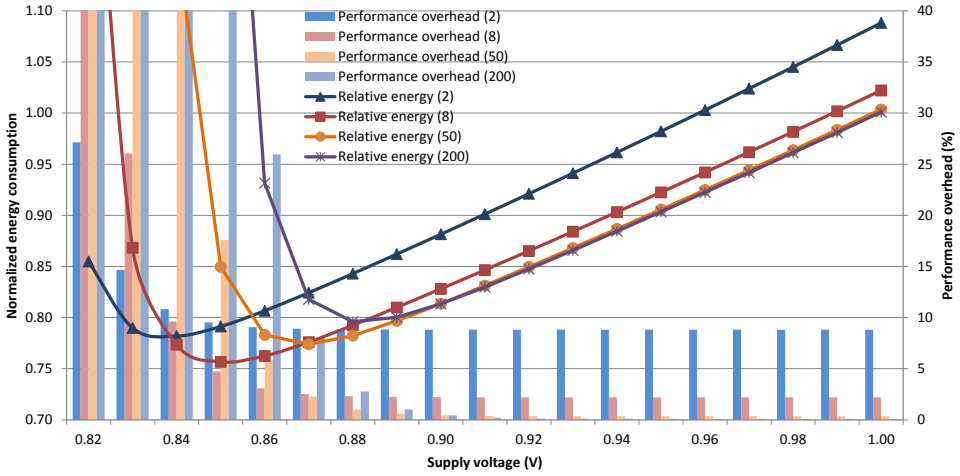


Figure 6.3: Optimal CD mapping and supply voltage for CP. Tuning the preservation interval.

in Listing 6.1 show a case where we choose a preservation interval of two loop iterations.

With our analytical model, we can estimate the energy efficiency of many possible preservation intervals for a given error rate. Figure 6.3 shows the analytical model results for four different preservation intervals. We can clearly see that longer preservation intervals (50, 200) perform better when error rates are low, and shorter preservation intervals (2, 8) perform better when error rates are high. For this particular benchmark, we find that a preservation interval of eight running at 0.85V yields the best result in terms of energy consumption.

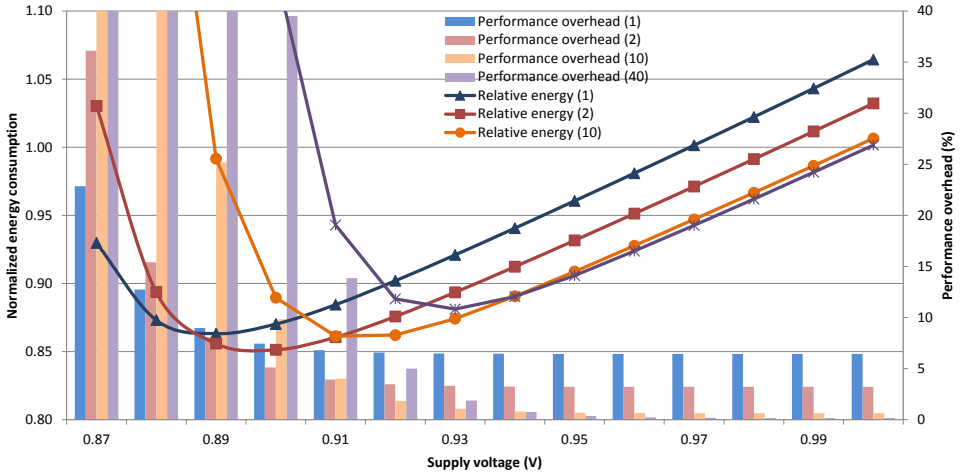


Figure 6.4: Optimal CD mapping and supply voltage for LIB. Each thread in the child CD preserves a float array with 80 elements.

6.1.4 LIB

LIB is similar to CP in that the main inner-loop body is a warp-scope CD that allows tuning the preservation interval. However, each thread in LIB needs to preserve a float array with 80 elements. Preserving this array into the shared memory greatly reduces the parallelism due to the limited capacity of the shared memory which in turn limits the number of thread blocks that can run concurrently. With a 4 byte float data type, each thread preserves 320 bytes, and since each thread block in LIB has 64 threads, we are preserving 20,480 bytes for each thread block, which means we can only schedule at most two thread blocks at a time rather than eight. Therefore, this array needs to be preserved in the global memory. Figure 6.4 shows that the overall energy efficiency is much lower for LIB due to the higher preservation overhead of using the global memory.

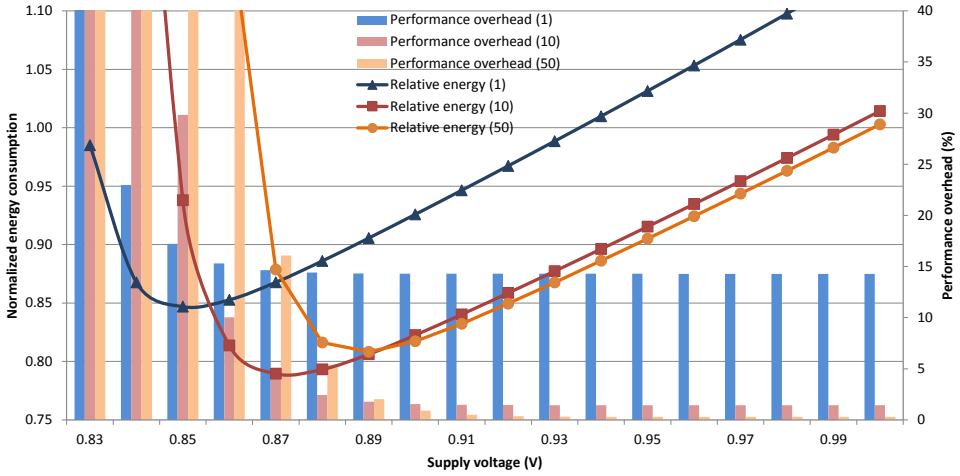


Figure 6.5: Optimal CD mapping and supply voltage for LPS. Thread-block-scope parent and child CDs preserving to shared memory.

6.1.5 LPS

LPS is similar to LIB in that the main loop consists of two nested CDs. However, both the parent and child CDs are thread-block-scope CDs with potentially higher recovery overhead, and preservation is done on a smaller data array (2448 bytes per thread block) in shared memory. As shown in Figure 6.5, the preservation overhead is still large enough to make CD mappings with a short preservation interval perform poorly. However, it is small enough to overcome the fact that all CDs are thread-block-scope, illustrated by LPS's better overall energy efficiency than LIB.

6.1.6 BFS and SSSP

For the purpose of tuning the preservation interval, it is generally better to have each thread go through many loop iterations so that we can try out

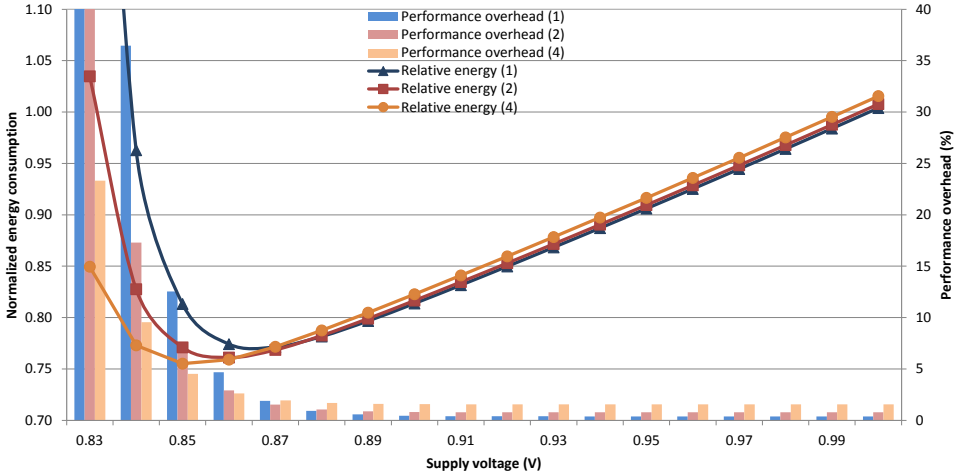


Figure 6.6: Optimal CD mapping and supply voltage for BFS. Each thread executes little amount of work while being irregular in terms of both control and data.

many different preservation intervals. However, some GPGPU workloads are written in a way that they launch a huge number of threads doing small amount of work; e.g. few loop iterations. BFS and SSSP are examples of such workloads. They are also unique in that they exhibit high degree of irregularity in both control and memory access patterns. In BFS and SSSP, each thread is assigned a node on the input graph and loops through all adjacent nodes to find a path that matches the objective of the function. In order to model these workloads, we examine how many neighbors each node has on average and use that number as the maximum preservation interval. Figure 6.6 and Figure 6.7 show the optimization space for BFS and SSSP. The average degree of the input graph we use is four and thus we try preservation intervals of one, two and four. While the graphs look very similar, SSSP suffers from higher

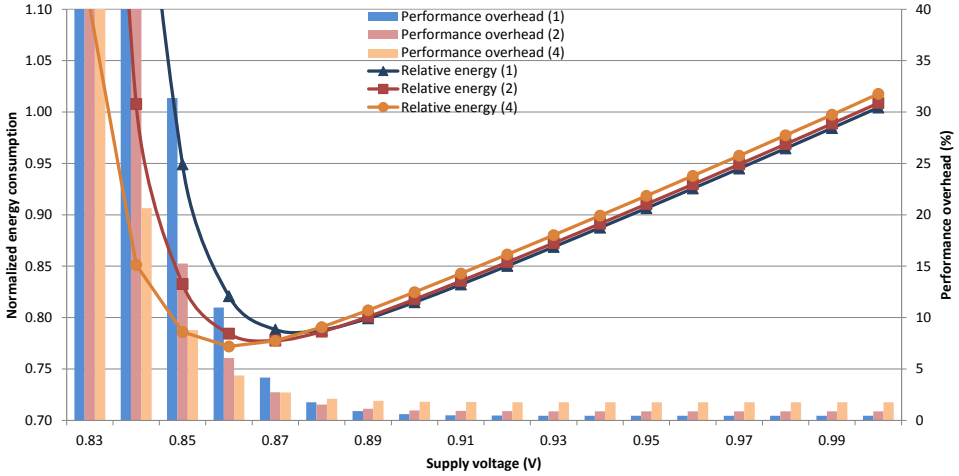


Figure 6.7: Optimal CD mapping and supply voltage for SSSP. Each thread executes little amount of work while being irregular in terms of both control and data.

recovery overhead at high error rates due to more irregular control and data patterns it exhibits during execution.

6.1.7 SPMV

The code structure of SPMV resembles that of CP. However, similarly to BFS and SSSP, SPMV also has relatively little work performed by each thread and has moderately irregular control and highly irregular data access patterns. The result is shown in Figure 6.8. Due to irregular data accesses, the memory access latency becomes the bottleneck and the preservation overhead becomes negligible. Thus, the smallest preservation interval (e.g. one) leads to the energy optimal operating point at 0.85V.

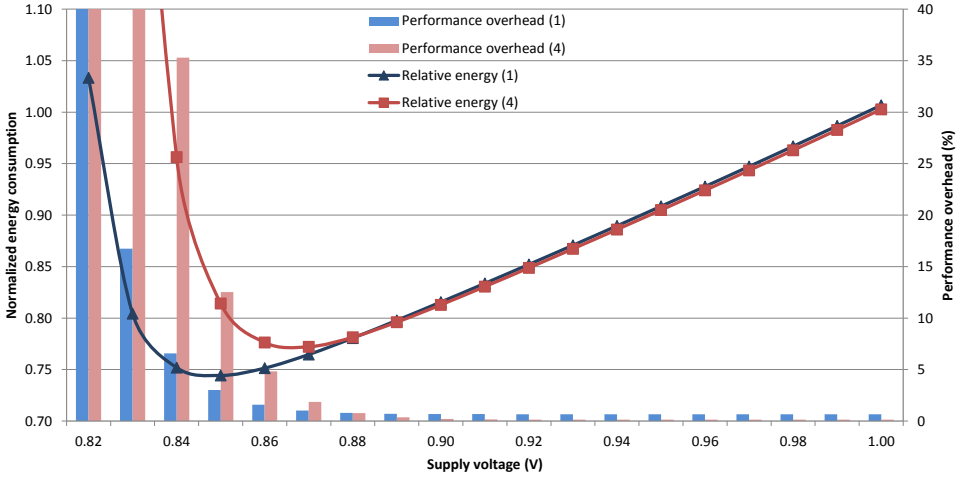


Figure 6.8: Optimal CD mapping and supply voltage for SPMV.

6.1.8 Model-Based Optimization Summary

Using our model-based optimization methodology, we have identified the energy optimal CD mappings and operating voltages for each of the benchmarks. Table 6.1 lists the energy optimal settings given by the analytical model. To account for modeling errors as discussed in Section 5.3.2, we also include several local optima and their adjacent points in the optimization space into our simulation based evaluation.

Table 6.1: Model-based optimization results

Name	**# nested CD levels**	**Preservation interval (loop iteration)**	**Voltage (V)**	**Error rate (errors/core /cycle)**	**Normalized energy**
AES	2	N/A	0.85	1.73E-5	0.82
LIB	2	2	0.90	9.66E-8	0.85
LPS	2	10	0.87	2.17E-6	0.79
CP	2	8	0.85	1.73E-5	0.76
SPMV	2	1	0.85	1.73E-5	0.74
BFS	2	4	0.85	1.73E-5	0.76
SSSP	2	4	0.86	6.12E-6	0.77
SP	3	N/A	0.81	1.09E-3	0.74

6.2 Detailed Analysis Using Cycle-Based Simulation

The results of the analytical model-based optimization show that depending on the kernel characteristics, FGCDs can help reduce the voltage margin significantly leading to energy savings of 25% or more. Now that we have identified the optimal CD mappings and their corresponding operating voltages of each benchmark, detailed analyses can be performed using the cycle-level simulation infrastructure described in Section 5.1.

6.2.1 Energy Savings and Performance Overheads

In this section, we show the energy savings of each benchmark at its energy optimal operating point. As discussed in Section 6.1.8, we simulate the optimal CD mappings in Table 6.1 as well as several local optima points given by the model. Figure 6.9 shows the results of the best performing points among all simulated candidates. In all cases except for SP, the optimal CD mapping and voltage given by the model is also the optimal CD mapping and voltage given by the simulator which conforms with the model validation results discussed in Section 5.3.2. For SP, the optimal voltage is one step higher than the model result: 0.82V as compared to 0.81V. This is mainly due to inaccuracies in profiling. SP runs the same kernel hundreds of times and each kernel run takes different amount of time in simulation while the model uses an average number for all kernel runs.

Figure 6.9 also shows the runtime overhead of each benchmark as a bar graph. Note that since we are optimizing for energy consumption, per-

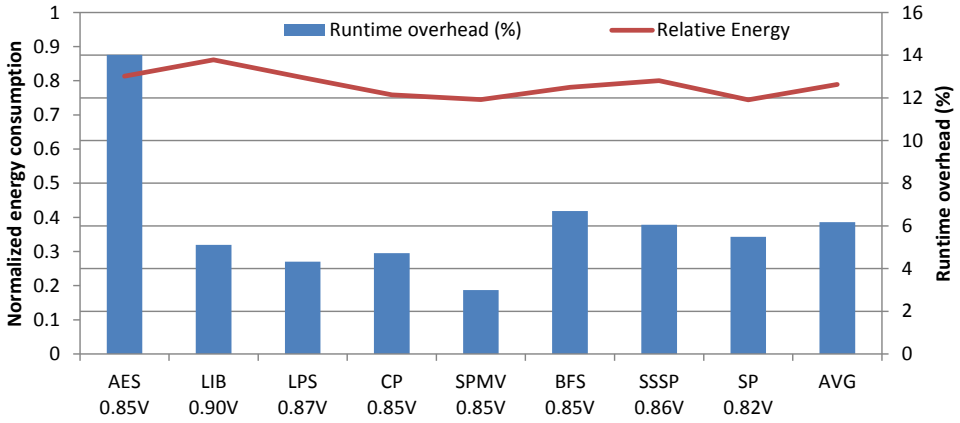


Figure 6.9: Simulation results for optimal CD mappings. Energy consumption normalized to the baseline architecture running at 1.0V without any error.

formance overhead of FGCDs can sometimes appear high. For example, AES exhibits 14% performance degradation due to the high preservation overhead that comes with the optimal two-level CD mapping.

Figure 6.10 shows the comparison between energy savings of FGCDs to other approaches. We use the analytical model to model other recovery schemes and compare their results to both the FGCDs model results and simulation results. Kernel restart simply re-executes the entire kernel in case of an error. Any mutable input states are preserved to make the kernel idempotent. Kernel restart works best for the AES kernel because the short duration of the kernel allows running the processor at a lower voltage. We also model the closest related work, iGPU [57]. Since there is no hierarchy in iGPU, it is similar to the case of using only the finest-grained CDs in our approach as shown in the figure. However, iGPU relies on the compiler analysis to identify short code regions with few live-in variables, and when no good candidates

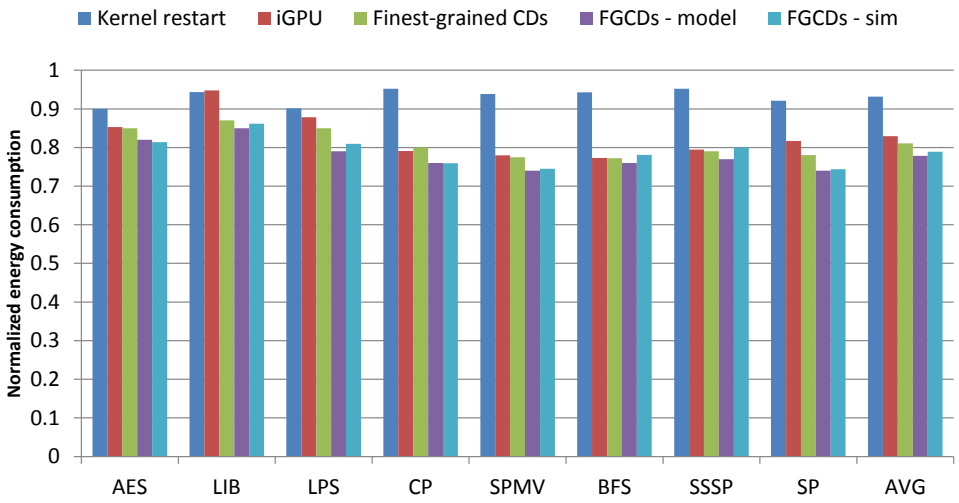


Figure 6.10: Comparison of FGCDs to other recovery schemes. Energy consumption normalized to the baseline architecture running at 1.0V without any error.

are found, 32 instructions are grouped to form an artificial idempotent region. As a result, iGPU slightly underperforms than Finest-grained CDs in most cases. Furthermore, as in the case of LPS and SP, iGPU may break idempotent regions prematurely when there are frequent store instructions, leading to idempotent regions that are too fine-grained. In general, the benefits of Finest-grained CDs over iGPU come from the API which allows us to take advantage of the application context available at the source code level. Lastly, Figure 6.10 also shows the comparison between the model and the simulation results for the optimal CD mappings, again validating our model-based optimization methodology.

Table 6.2: Combinations of error containment mechanisms

Abbreviation	**store instruction**	*cd.end* **instruction**
SS	Instruction stall	Instruction stall
SE	Instruction stall	CD escalation
BS	Store buffer	Instruction stall
BE	Store buffer	CD escalation

6.2.2 Sensitivity to Error Detection Latency

In our evaluation, we assume that sensor-based voltage droop detectors are available at the granularity of SIMD processors. Depending on the design and the placement, these error detectors can have different error detection latencies ranging from zero to a few tens of cycles. In this section, we run sensitivity studies on error detection latency for three error containment mechanisms discussed in Section 4.4.2: instruction stall, store buffer, and CD escalation. As previously discussed, two types of instructions have the potential to violate error containment of a CD: store instructions and *cd.end* instructions. First, store instructions can be either stalled or buffered with the store buffer to ensure no erroneous state is propagated outside of the CD. Note that CD escalation cannot be used since store instructions can write to unintended addresses due to errors corrupting preserved state of the parent. Second, *cd.end* instructions can be either stalled or speculatively executed with CD escalation taking care of rare misspeculations. Each row in Table 6.2 shows a valid combination of error containment mechanisms that we evaluate.

Figure 6.11 illustrates the sensitivities of SS and SE to error detection latency. The optimal CD mappings and their corresponding voltages as listed

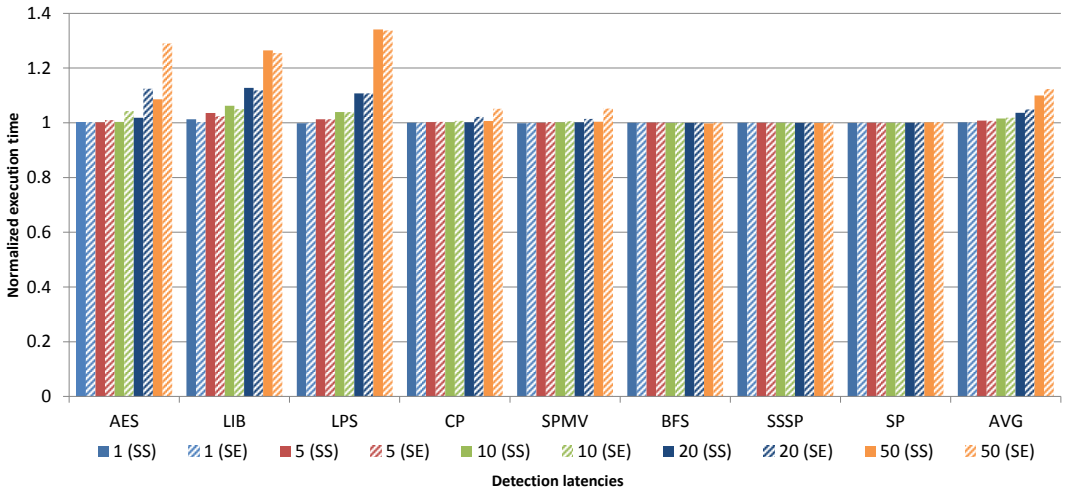


Figure 6.11: Sensitivity of SS and SE to error detection latency. Execution time normalized to the baseline of zero cycle error detection.

in Table 6.1 are used for simulation. Note that the optimal CD mappings used for simulation assume zero cycle error detection latency and they may not be optimal when there is latency in detection. We choose these CD mappings as a reasonable simulation target for studying the sensitivity of different error containment mechanisms to error detection latency.

The y-axis in Figure 6.11 represents the execution time of each benchmark normalized to the baseline of zero-cycle error detection latency. For both SS and SE, benchmarks CP, SPMV, BFS, SSSP, and SP appear insensitive to error detection latency of up to 50 cycles. This reaffirms that GPUs' massive multithreading is very effective at hiding latencies. On the other hand, AES, LIB, and LPS are more sensitive to error detection latency because these benchmarks issue bursts of store instructions either during preservation or computation exceeding the latency tolerance limit of the simulated GPU

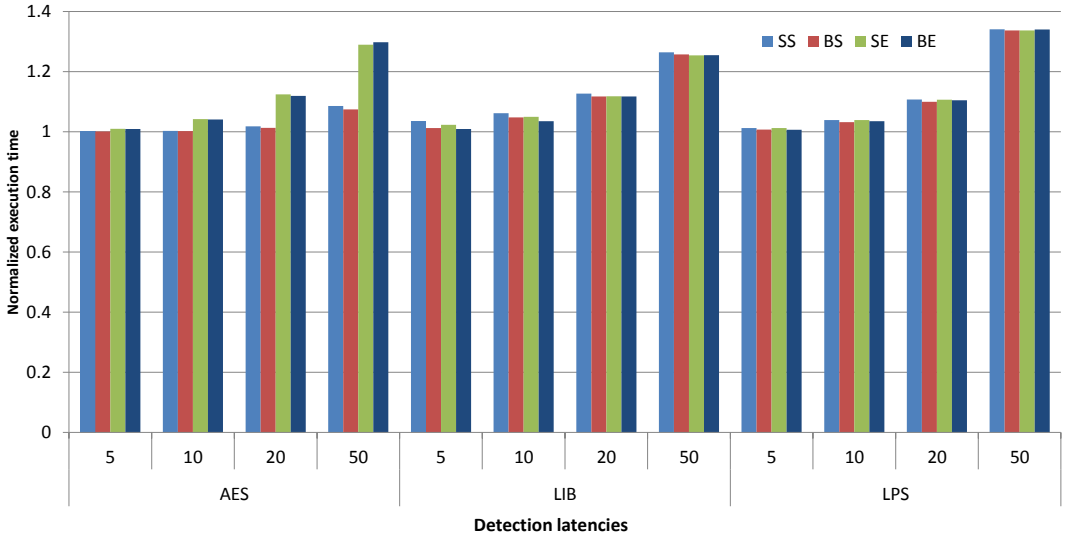


Figure 6.12: Comparison of four error containment mechanisms for store-intensive benchmarks. Execution time normalized to the baseline of zero cycle error detection.

architecture. Another interesting observation is that the differences between SS and SE are small. This means that the performance hit of these mechanisms is mainly from stalled store instructions rather than *cd.end* instructions for most benchmarks. AES shows a different trend because the kernel has very short child CDs having only a few tens of instructions. With such short CDs, the probability of escalation quickly reaches 1.0 as we increase the error detection latency and the performance hit of SE becomes much higher than SS due to its higher recovery overhead. A similar trend can be observed for CP and SPMV as well, although the impact on performance is smaller. On the contrary, LIB has a much longer child CD containing 4,000 instructions and a higher operating voltage (0.90V) leading to very infrequent escalation.

As shown in Figure 6.11, stalling store instructions can lead to noticeable performance degradation for benchmarks that write heavily to memory. An alternative way of ensuring error containment for store instructions is through the use of store buffers as explained in Section 4.4.2.2. Store buffers allow warps to speculatively execute past the store instruction without stalling. Figure 6.12 shows comparisons between stalling stores (SS, SE) and buffering stores (BS, BE) We use a store buffer size of 2KB as discussed in Section 4.4.2.2, and run sensitivity studies for AES, LIB and LPS benchmarks which suffer from performance degradation caused by stalled store instructions. Interestingly, the simulation results shown in Figure 6.12 illustrate that the benefits of adding a store buffer are only marginal. This is mainly due to the fact that these benchmarks issue bursts of store instructions quickly filling up the store buffer. Since the store buffer can hold up to 8 back-to-back store instructions, some gain is observed for error detection latencies of 5 and 10 cycles. However, the benefit of store buffer diminishes quickly as we further increase the detection latency. In summary, simple stall based error containment mechanisms work reasonably well at short error detection latencies of up to 10 to 20 cycles. Beyond that, none of the approaches are effective in keeping the performance overhead low.

Note that the sensitivity studies shown in Figure 6.11 and Figure 6.12 use a CD mapping that is optimized for zero cycle detection latency and may not be optimal in the presence of detection latency.

Chapter 7

Conclusions

As the process technology continues to scale, conventional worst case based designs will become increasingly inefficient due to excessive design margins required to guarantee correct operation. Timing speculation has emerged as a viable alternative design strategy that can optimize for the typical case by allowing errors to occur and efficiently recovering from those errors. This dissertation presents a framework that enables timing speculation on throughput processors such as GPUs. I first discuss why existing timing speculation techniques designed for CPUs cannot be directly applied to GPUs, and propose a hardware/software co-design approach to address the issues. With a combination of analytical model and cycle-base simulation, I demonstrate that the proposed scheme can substantially improve the efficiency of GPUs by running a program at its optimal operating point for the typical case. To continue to enjoy the benefit of technology scaling, I believe a new design strategy such as the proposed scheme will be essential in addressing the problem of design margins. Below is a summary of my key contributions.

I develop a set of hardware and software techniques to provide efficient state preservation, restoration and recovery on GPUs. The proposed API lets

the programmer take advantage of high-level knowledge available at the source code level and describe the key CD properties of the kernel which can further be analyzed and tuned using the analytical model discussed in Section 4.2.3. Unlike previous proposals, my approach does not require a specific type of error detector, does not have to checkpoint the entire thread context, and can trade-off between preservation and recovery overhead to achieve the best efficiency enabling timing speculation to be effective for a wider range of error rates and applications.

Bibliography

[1] Jinsuk Chung, Ikhwan Lee, Michael Sullivan, Jee Ho Ryoo, Dong Wan Kim, Doe Hyun Yoon, Larry Kaplan, and Mattan Erez. Containment domains: A scalable, efficient, and flexible resilience scheme for exascale systems. In *the Proceedings of SC12*, November 2012.

[2] Smitha Shyam, Kypros Constantinides, Sujay Phadke, Valeria Bertacco, and Todd Austin. Ultra low-cost defect protection for microprocessor pipelines. In *ACM Sigplan Notices*, volume 41, pages 73–82. ACM, 2006.

[3] Meeta S Gupta, Krishna K Rangan, Michael D Smith, Gu-Yeon Wei, and David Brooks. Decor: A delayed commit and rollback mechanism for handling inductive noise in processors. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 381–392. IEEE, 2008.

[4] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Fermi, 2009.

[5] NVIDIA Corporation. Whitepaper: NVIDIA GeForce GTX 680, 2012.

[6] AMD Corporation. AMD Radeon HD 7900 Series Specification, 2011.

[7] Intel Corporation. Intel HD Graphics OpenSource Programmer Reference Manual, June 2011.

[8] NVIDIA Corporation. NVIDIA CUDA Programming Guide, 2011.

[9] AMD Corporation. ATI Stream Computing OpenCL Programming Guide, August 2010.

[10] Collange, Sylvain. Stack-less SIMT Reconvergence At Low Cost, 2011.

[11] Wilson W.L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *40th International Symposium on Microarchitecture (MICRO-40)*, December 2007.

[12] ITRS 2011 Edition, Design. *International Technology Roadmap for Semiconductors*, 2011.

[13] Vijay Janapa Reddi, Svilen Kanev, Wonyoung Kim, Simone Campanoni, Michael D Smith, Gu-yeon Wei, and David Brooks. Voltage smoothing: Characterizing and mitigating voltage noise in production processors via software-guided thread scheduling. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 77–88. IEEE, 2010.

[14] N James, Ph Restle, J Friedrich, B Huott, and B McCredie. Comparison of split-versus onnected-core supplies in the POWER6 microprocessor. In *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pages 298–604. IEEE, 2007.

[15] Todd Austin, Valeria Bertacco, David Blaauw, and Trevor Mudge. Opportunities and challenges for better than worst-case design. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, pages 2–7. ACM, 2005.

[16] Intel Corporation. Power and Thermal Management in the Intel® Core TM. *Intel® Centrino® Duo Mobile Technology*, 10(2):109, 2006.

[17] Hector Sanchez, Belli Kuttanna, Tim Olson, Mike Alexander, Gian Gerosa, Ross Philip, and Jose Alvarez. Thermal management system for high performance PowerPC Microprocessors. In *Compcon'97. Proceedings, IEEE*, pages 325–330. IEEE, 1997.

[18] Russ Joseph, David Brooks, and Margaret Mart/onosi. Control techniques to eliminate voltage emergencies in high performance processors. In *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*, pages 79–90. IEEE, 2003.

[19] Michael D Powell and TN Vijaykumar. Exploiting resonant behavior to reduce inductive noise. In *ACM SIGARCH Computer Architecture News*, volume 32, page 288. IEEE Computer Society, 2004.

[20] Ed Grochowski, Dave Ayers, and Vivek Tiwari. Microarchitectural simulation and control of di/dt-induced power supply voltage variation. In *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*, pages 7–16. IEEE, 2002.

[21] Alan Drake, Robert Senger, Harmander Deogun, Gary Carpenter, Soraya Ghiasi, Tuyet Nguyen, Norman James, Michael Floyd, and Vikas Pokala. A distributed critical-path timing monitor for a 65nm high-performance microprocessor. In *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pages 398–399. IEEE, 2007.

[22] Charles R Lefurgy, Alan J Drake, Michael S Floyd, Malcolm S Allen-Ware, Bishop Brock, Jose A Tierno, and John B Carter. Active management of timing guardband to save energy in POWER7. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1–11. ACM, 2011.

[23] James Tschanz, Keith Bowman, Steve Walstra, Marty Agostinelli, Tanay Karnik, and Vivek De. Tunable replica circuits and adaptive voltage-frequency techniques for dynamic voltage, temperature, and aging variation tolerance. In *2009 Symposium on VLSI Circuits*, 2009.

[24] Vijay Janapa Reddi, Meeta Sharma Gupta, Glenn Holloway, Gu-Yeon Wei, Michael D Smith, and David Brooks. Voltage emergency prediction: Using signatures to reduce operating margins. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 18–29. IEEE, 2009.

[25] Dan Ernst, Nam Sung Kim, Shidhartha Das, Sanjay Pant, Rajeev Rao, Toan Pham, Conrad Ziesler, David Blaauw, Todd Austin, Krisztian Flaut-

ner, et al. Razor: A low-power pipeline based on circuit-level timing speculation. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 7–18. IEEE, 2003.

[26] David Blaauw, Sudherssen Kalaiselvan, Kevin Lai, Wei-Hsiang Ma, Sanjay Pant, Carlos Tokunaga, Shidhartha Das, and David Bull. Razor II: In situ error detection and correction for PVT and SER tolerance. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pages 400–622. IEEE, 2008.

[27] Evgeni Krimer, Patrick Chiang, and Mattan Erez. Lane decoupling for improving the timing-error resiliency of wide-simd architectures. In *Proceedings of the 39th International Symposium on Computer Architecture*, pages 237–248. IEEE Press, 2012.

[28] Keith A Bowman, James W Tschanz, Nam Sung Kim, Janice C Lee, Chris B Wilkerson, S-LL Lu, Tanay Karnik, and Vivek K De. Energy-efficient and metastability-immune resilient circuits for dynamic variation tolerance. *Solid-State Circuits, IEEE Journal of*, 44(1):49–63, 2009.

[29] Jing Xin and Russ Joseph. Identifying and predicting timing-critical instructions to boost timing speculation. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 128–139. ACM, 2011.

[30] Haitham Akkary, Ravi Rajwar, and Srikanth T Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window pro-

cessors. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 423–434. IEEE, 2003.

[31] José F Martínez, Jose Renau, Michael C Huang, and Milos Prvulovic. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *Microarchitecture, 2002.(MICRO-35). Proceedings. 35th Annual IEEE/ACM International Symposium on*, pages 3–14. IEEE, 2002.

[32] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced cpu energy. In *Mobile Computing*, pages 449–471. Springer, 1996.

[33] Jason Duell. The design and implementation of Berkeley lab's Linux checkpoint/restart. *Lawrence Berkeley National Laboratory*, 2005.

[34] Amnon Barak, Shai Guday, and Richard G Wheeler. *The MOSIX distributed operating system: load balancing for UNIX*. Springer-Verlag New York, Inc., 1993.

[35] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. *Checkpoint and migration of UNIX processes in the Condor distributed processing system*. Computer Sciences Department, University of Wisconsin, 1997.

[36] James S Plank, Yuqun Chen, Kai Li, Micah Beck, and Gerry Kingsley. Memory exclusion: Optimizing the performance of checkpointing systems. *Software-Practice and Experience*, 29(2):125–142, 1999.

[37] Adam Beguelin, Erik Seligman, and Peter Stephan. Application level fault tolerance in heterogeneous networks of workstations. *Journal of Parallel and Distributed Computing*, 43(2):147–155, 1997.

[38] John Paul Walters and Vipin Chaudhary. Application-level checkpointing techniques for parallel programs. In *Distributed Computing and Internet Technology*, pages 221–234. Springer, 2006.

[39] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Automated application-level checkpointing of MPI programs. *ACM Sigplan Notices*, 38(10):84–94, 2003.

[40] Greg Bronevetsky, Daniel Marques, Keshav Pingali, Peter Szwed, and Martin Schulz. Application-level checkpointing for shared memory programs. *ACM SIGOPS Operating Systems Review*, 38(5):235–247, 2004.

[41] Nitin H Vaidya. A case for two-level distributed recovery schemes. In *ACM SIGMETRICS Performance Evaluation Review*, volume 23, pages 64–73. ACM, 1995.

[42] BS Panda and Sajal K Das. Performance evaluation of a two level error recovery scheme for distributed systems. In *Distributed Computing*, pages 88–97. Springer, 2002.

[43] Milos Prvulovic, Zheng Zhang, and Josep Torrellas. ReVive: cost-effective architectural support for rollback recovery in shared-memory multiproces-

sors. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, pages 111–122. IEEE, 2002.

[44] Daniel J Sorin, Milo MK Martin, Mark D Hill, and David A Wood. SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, pages 123–134. IEEE, 2002.

[45] Brian Randell. System structure for software fault tolerance. In *ACM SIGPLAN Notices*, volume 10, pages 437–449. ACM, 1975.

[46] E.N. Elnozahy and W. Zwaenepoel. Manetho: transparent roll back-recovery with low overhead, limited rollback, and fast output commit. *Computers, IEEE Transactions on*, 41(5):526 –531, May 1992.

[47] B. H. L. Alvisi and K. Marzullo. Nonblocking and orpha-free message logging protocols. In *Proc. IEEE Fault Tolerant Computing Symp. (FTCS)*.

[48] K. Li, J. F. Naughton, and J. S. Plank. Checkpointing multicomputer applications. In *Proc. IEEE Symp. Reliable Distr. Syst.*, 1991.

[49] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, Jan. 1987.

[50] Patrick J Meaney, Scott B Swaney, Pia N Sanda, and Lisa Spainhower. IBM z990 soft error detection and recovery. *Device and Materials Reliability, IEEE Transactions on*, 5(3):419–427, 2005.

[51] Maurice Herlihy and J Eliot B Moss. *Transactional memory: Architectural support for lock-free data structures*, volume 21. ACM, 1993.

[52] Lance Hammond, Vicky Wong, Mike Chen, Brian D Carlstrom, John D Davis, Ben Hertzberg, Manohar K Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *ACM SIGARCH Computer Architecture News*, volume 32, page 102. IEEE Computer Society, 2004.

[53] Kevin E Moore, Jayaram Bobba, Michelle J Moravan, Mark D Hill, David A Wood, et al. LogTM: log-based transactional memory. In *HPCA*, volume 6, pages 254–265, 2006.

[54] Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. Relax: An architectural framework for software recovery of hardware faults. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 497–508. ACM, 2010.

[55] Gulay Yalcin, Osman Unsal, Ibrahim Hur, Adrian Cristal, Mateo Valero, et al. FaulTM: Fault-tolerance using hardware transactional memory. In *Pespma 2010-Workshop on Parallel Execution of Sequential Programs on Multi-core Architecture*, 2010.

[56] Shuguang Feng, Shantanu Gupta, Amin Ansari, Scott A Mahlke, and David I August. Encore: Low-cost, fine-grained transient fault recovery. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 398–409. ACM, 2011.

[57] Jaikrishnan Menon, Marc De Kruijf, and Karthikeyan Sankaralingam. iGPU: exception support and speculative execution on GPUs. In *Proceedings of the 39th International Symposium on Computer Architecture*, pages 72–83. IEEE Press, 2012.

[58] Marc de Kruijf and Karthikeyan Sankaralingam. Idempotent processor architecture. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 140–151. ACM, 2011.

[59] Gaurang Upasani, Xavier Vera, and Antonio Gonzalez. Avoiding core's DUE & SDC via acoustic wave detectors and tailored error containment and recovery. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 37–48. IEEE, 2014.

[60] Charles E Molnar, Robert F Sproull, and Ivan E Sutherland. Counterflow pipeline processor architecture. 1994.

[61] Vijay Balasubramanian and Prithviraj Banerjee. Compiler-assisted synthesis of algorithm-based checking in multiprocessors. *IEEE Transactions on Computers*, (4):436–446, 1990.

[62] Ahmad Al-Yamani, Nahmsuk Oh, Edward J McCluskey, et al. Performance evaluation of checksum-based abft. In *Defect and Fault Tolerance in VLSI Systems, 2001. Proceedings. 2001 IEEE International Symposium on*, pages 461–466. IEEE, 2001.

[63] Hyungmin Cho, Larkhoon Leem, and Subhasish Mitra. Ersa: Error resilient system architecture for probabilistic applications. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 31(4):546–558, 2012.

[64] Vaibhav Gupta, Debabrata Mohapatra, Anand Raghunathan, and Kaushik Roy. Low-power digital signal processing using approximate adders. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 32(1):124–137, 2013.

[65] Shih-Lien Lu. Speeding up processing with approximation circuits. *Computer*, 37(3):67–73, 2004.

[66] Jiawei Huang, John Lach, and Gabriel Robins. A methodology for energy-quality tradeoff using imprecise hardware. In *Proceedings of the 49th Annual Design Automation Conference*, pages 504–509. ACM, 2012.

[67] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Architecture support for disciplined approximate programming. In *ACM SIGPLAN Notices*, volume 47, pages 301–312. ACM, 2012.

[68] Jeffrey T Ludwig, S Hamid Nawab, and Anantha P Chandrakasan. Low-power digital filtering using approximate processing. *Solid-State Circuits, IEEE Journal of*, 31(3):395–400, 1996.

[69] S Hamid Nawab, Alan V Oppenheim, Anantha P Chandrakasan, Joseph M Winograd, and Jeffrey T Ludwig. Approximate signal pro-

cessing. *Journal of VLSI signal processing systems for signal, image and video technology*, 15(1-2):177–200, 1997.

[70] Michael Carbin, Sasa Misailovic, and Martin C Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *ACM SIGPLAN Notices*, volume 48, pages 33–52. ACM, 2013.

[71] Anys Bacha and Radu Teodorescu. Dynamic Reduction of Voltage Margins by Leveraging On-chip ECC in Itanium II Processors. In *International Symposium on Computer Architecture (ISCA)*, pages 1–11, 2013.

[72] Kim Hazelwood and David Brooks. Eliminating voltage emergencies via microarchitectural voltage control feedback and dynamic optimization. In *Proceedings of the 2004 international symposium on Low power electronics and design*, pages 326–331. ACM, 2004.

[73] Meeta S Gupta, Jarod L Oatley, Russ Joseph, Gu-Yeon Wei, and David M Brooks. Understanding voltage variations in chip multiprocessors using a distributed power-delivery network. In *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE'07*, pages 1–6. IEEE, 2007.

[74] Vijay Janapa Reddi, Meeta S Gupta, Michael D Smith, Gu-yeon Wei, David Brooks, and Simone Campanoni. Software-assisted hardware reliability: abstracting circuit-level challenges to the software stack. In *Proceedings of the 46th Annual Design Automation Conference*, pages 788–793. ACM, 2009.

[75] Jingwen Leng, Yazhou Zu, Minsoo Rhu, Meeta Gupta, and Vijay Janapa Reddi. GPUVolt: modeling and characterizing voltage noise in GPU architectures. In *Proceedings of the 2014 international symposium on Low power electronics and design*, pages 141–146. ACM, 2014.

[76] Wonyoung Kim, Meeta S Gupta, Gu-Yeon Wei, and David Brooks. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 123–134. IEEE, 2008.

[77] Andrew Adinetz. Adaptive Parallel Computation with CUDA Dynamic Parallelism. http://devblogs.nvidia.com/parallelforall/introduction-cuda-dynamic-parallelism.

[78] NVIDIA Corporation. GPU Computing SDK. https://developer.nvidia.com/gpu-computing-sdk.

[79] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and WMW Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 2012.

[80] NVIDIA Corporation. PTX: Parallel Thread Execution ISA Version 2.3, 2011.

[81] Markus Steinberger, Michael Kenzel, Bernhard Kainz, and Dieter Schmalstieg. Scatteralloc: Massively parallel dynamic memory allocation for the

GPU. In *Innovative Parallel Computing (InPar), 2012*, pages 1–10. IEEE, 2012.

[82] AMD Corp. BIOD and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors, 2012.

[83] IBM Corp. Power7 System RAS – Key Aspects of Power Systems Reliability, Availability, and Serviceability. `http://www-03.ibm.com/systems/power/hardware/whitepapers/ras7.html`, 2012.

[84] Intel Corp. OS Machine Check Recovery on Itanium-Based Systems. `http://www.intel.com/content/dam/www/public/us/en/documents/application-notes/os-machine-check-recovery-itanium-application-note.pdf`, 2008.

[85] Intel Corp. Intel Itanium Processor Family Error Handling Guide. `http://www.intel.com/content/www/us/en/processors/itanium/itanium-error-handling-guide.html`, 2010.

[86] Timothy Rogers, Mike O'Connor, and Tor Aamodt. Cache-Conscious Wavefront Scheduling. In *45th International Symposium on Microarchitecture (MICRO-45)*, December 2012.

[87] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. Analyzing CUDA workloads using a detailed GPU simulator.

In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163–174. IEEE, 2009.

[88] GPGPU-Sim Manual. http://www.gpgpu-sim.org/manual.

[89] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A quantitative study of irregular programs on GPUs. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, pages 141–151. IEEE, 2012.

[90] Svetlin A Manavski. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. In *Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on*, pages 65–68. IEEE, 2007.

[91] Mike Giles and Su Xiaoke. Notes on using the NVIDIA 8800 GTX graphics card, 2008.

[92] Alfredo Braunstein, Marc Mézard, and Riccardo Zecchina. Survey propagation: An algorithm for satisfiability. *Random Structures & Algorithms*, 27(2):201–226, 2005.

[93] Alain J Martin. Towards an energy complexity of computation. *Information Processing Letters*, 77(2):181–187, 2001.